VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Použití knihovny LibUSB v Linuxu pro komunikaci s mikropočítačem přes USB rozhraní

Library LibUSB for Communication with Microcontroller over USB

2012 Bc. Michal Kolář

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michal Kolář**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Použití knihovny LibUSB v Linuxu pro komunikaci s mikropočítačem přes USB rozhraní
Library LibUSB for Communication with Microcontroller over USB

Zásady pro vypracování:

Navrhněte a vyzkoušejte knihovnu LibUSB pro obousměrnou komunikaci přes USB s libovolným mikropočítačem PIC řady 18. Komunikace musí umožňovat nejen režim dotaz-odpoveď, ale i automatické vysílání dat z mikropočítače a naopak - zpracování událostí, přerušení a pod.
1. Seznamte se s vlastnostmi mikropočítačů PIC řady 18 s USB rozhraním. Vyberte vhodný typ. Popiště jednotlivé pracovní režimy, nastavení rychlostí, identifikace a pod.
2. Popiště knihovnu LibUSB pro Linux. Logiku programování, konfiguraci, ovladače, základní funkce, identifikaci zařízení.
3. Navrhněte programové vybavení v jazyce C pro mikrokontrolér tak, aby bylo možno obousměrně komunikovat způsobem dotaz-odpověď a asynchronně.
4. Navrhněte odpovídající programové řešení s pomocí knihovny LibUSB, aby byly splněny stejné požadavky, jako pro program mikrokontroléru. Řešení navrhněte v jazyce C/C++ bez závislostí na konkrétním překladači.
5. Proveďte testování stability, spolehlivosti, rychlosti, odezvy i komunikaci s více mikroprocesory.

Seznam doporučené odborné literatury:

Kainka, Burkhard: USB - Měření, řízení a regulace pomocí sběrnice USB, BEN - technická literatura, ISBN 80-7300-073-3
http://www.libusb.org/
http://www.microchip.com - technické listy

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Olivka**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23.4.2012 .....

.....

Rád bych poděkoval panu Ing. Petrovi Olivkovi za odbornou pomoc a konzultaci při vytváření této diplomové práce.

Abstrakt

Tato diplomová práce se zabývá vytvořením demonstrační aplikace, která ověřuje funkcionalitu volně dostupné a šiřitelné knihovny LibUSB v Linuxu při obousměrné komunikaci s libovolným mikrokontrolérem PIC řady 18 přes USB rozhraní. Součástí je také návod, jak navrhnout program mikrokontroléru (firmware) s použitím volně dostupných balíků firmy Microchip. Celou diplomovou práci lze pojmout také jako doprovodný materiál při výuce. Všechny zdrojové kódy jsou napsány a okomentovány v anglickém jazyce, čímž se stávají srozumitelné pro zahraniční studenty.

Klíčová slova: LibUSB, USB, mikrokontrolér, PIC, PIC18F26J50, Microchip, synchronní komunikace, asynchronní komunikace

Abstract

This diploma thesis is concerned with creating a demo application that verifies the functionality of the open source library LibUSB in Linux for bidirectional communication with any 18 series PIC microcontroller through the USB interface. It also includes instructions on how to design a microcontroller firmware using the Microchip freeware packages. The entire thesis can be understood as accompanying educational material. All source code is written and commented in English, making it understandable for foreign students.

Key Words: LibUSB, USB, microcontroller, PIC, PIC18F26J50, Microchip, synchronous communication, asynchronous communication

Seznam použitých symbolů a zkratek

Zkratka		Popis
1-Wire	–	Systém sériové komunikace využívající jeden vodič, který na rozdíl od sběrnice I^2C nabízí menší přenosovou rychlost a větší dosah.
API	–	Angl. Application Programming Interface, označuje v informatice rozhraní pro programování aplikací. Jde o sbírku procedur, funkcí či tříd nějaké knihovny (ale třeba i jiného programu nebo jádra operačního systému), které může programátor využívat. API určuje, jakým způsobem jsou funkce knihovny volány ze zdrojového kódu programu.
CPU	–	Procesor nebo mikroprocesor (angl. Central Processing Unit) je v informatice základní součást počítače, která vykonává strojový kód spuštěného počítačového programu.
EEPROM	–	Také E^2PROM (angl. Electrically Erasable Programmable Read-Only Memory je elektricky mazatelná semipermanentní (nevolatilní) paměť typu ROM-RAM s omezenějším počtem zápisů než má paměť typu flash a před novým naprogramováním je nutné ji nejprve celou smazat.
Firmware	–	V elektronice a výpočetní technice se často používá k označení něčeho stálého, většinou poměrně malého, např. programy nebo datové struktury, které vnitřně ovládají různá elektronická zařízení.
FLASH	–	Flash paměť (nebo jen flash) je nevolatilní (semipermanentní) elektricky programovatelná (zapisovatelná) paměť s libovolným přístupem. Paměť je vnitřně organizována po blocích a na rozdíl od pamětí typu EEPROM, lze programovat každý blok samostatně (obsah ostatních bloků je zachován).
I^2C	–	Angl. Inter-Integrated Circuit, je počítačová sériová sběrnice vyvinutá firmou Philips, která je používána k připojování nízkorychlostních periferií k základní desce, vestavěnému systému, mobilnímu telefonu atd.
ID	–	Zkratka pro identifikátor, sloužícího jako jednoznačná identifikace jednotlivých entit.
IDE	–	Vývojové prostředí (angl. Integrated Development Environment) je software usnadňující práci programátorů, většinou zaměřené na jeden konkrétní programovací jazyk. Obsahuje editor zdrojového kódu, kompilátor, interpret, ladící modul (angl. debugger) atd.

LED – Angl. Light-Emitting Diode, je dioda emitující světlo. Je to elektronická polovodičová součástka obsahující přechod P-N. Na rozdíl od klasických diod, LED vyzařuje viditelné, infračervené, případně ultrafialové světlo v úzkém spektru barev.

Mikrokontrolér – Angl. microcontroller, nebo také MCU, µC, uC je většinou monolitický integrovaný obvod obsahující kompletní mikropočítač. Vyznačuje se velkou spolehlivostí a kompaktností a používá se především pro jednoúčelové aplikace jako je řízení, regulace apod. Více informací lze najít na internetových stránkách [1, 2].

MIPS – Milion instrukcí za sekundu (angl. Million Instruction Per Second). Je to jednotka výkonnosti počítače, která udává počet zpracovaných instrukcí za sekundu. Alternativním označením je MOPS - milion operací za sekundu (angl. Million Operations Per Second).

MSSP – Angl. Master Synchronous Serial Port, je sériové rozhraní pro komunikaci s jinými periferiemi (např. sériové EEPROM, posuvné registry, ovladače displejů, A/D převodníky atd.) nebo mikrokontroléry. Může pracovat v režimu SPI, nebo I^2C.

OS – Operační systém (angl. Operating System) je v informatice základní programové vybavení počítače (tj. software), které je zavedeno do paměti počítače při jeho startu a zůstává v činnosti až do jeho vypnutí. Skládá se z jádra (kernel) a pomocných systémových nástrojů. Hlavním úkolem operačního systému je zajistit uživateli možnost ovládat počítač, vytvořit pro procesy stabilní aplikační rozhraní (API) a přidělovat jim systémové zdroje.

OTP – Anglicky One-Time Programmable, je permanentní paměť typu ROM, která je jednorázově programovatelná.

PC – Osobní počítač (angl. Personal Computer) je označení počítače určeného pro použití jednotlivcem (na rozdíl od dřívějších střediskových počítačů resp. sálových počítačů).

PIC – Mikrokontroléry PIC jsou programovatelné polovodičové součástky – jednočipové mikropočítače (mikrořadiče, mikrokontroléry) vyráběné firmou Microchip Technology sídlící v USA. Podle různých zdrojů zkratka PIC v angličtině znamená: Programmable Interface Controller, Programmable Intelligent Computer, nebo Peripheral Interface Controller.

PLL – Angl. Phase-locked loop, nebo Phase Lock Loop je řídící systém, který generuje výstupní signál, jehož fáze je odvozená od fáze vstupního referenčního signálu. Je to elektronický obvod, který se skládá z proměnného frekvenčního oscilátoru a fázového detektoru. Porovnává fázi vstupního signálu s fází odvozeného signálu z výstupního oscilátoru a upravuje frekvenci, aby si fáze odpovídali.

RAM – Paměť s přímým přístupem (angl. Random-Access Memory) je typ elektronické paměti, která umožňuje přístup k libovolné části v konstantním čase bez ohledu na její fyzické umístění (na rozdíl od sekvenčních pamětí jako je magnetická páska, optický disk nebo pevný disk).

RISC – Anglicky Reduced Instruction Set Computer, je počítač s redukovanou instrukční sadou, takže složitější instrukce se musejí implementovat jako sekvence těch existujících. Délka provádění jedné instrukce je vždy jeden cyklus.

ROM – Paměť pouze pro čtení (angl. Read-Only Memory) je typ elektronické paměti, jejíž obsah je dán při výrobě, není závislý na napájení (je tzv. nevolatilní) a nelze ji později přepsat. Používá se pro uložení firmware v elektronických zařízeních. V minulosti byly paměti typu ROM v počítačích používány pro uložení BIOSu.

RS232 – Standard RS-232, resp. jeho poslední varianta RS-232C z roku 1969, nebo také sériový port nebo sériová linka se používá jako komunikační rozhraní osobních počítačů a další elektroniky. Umožňuje propojení a vzájemnou sériovou komunikaci dvou zařízení po jednom páru vodičů v každém směru. Na rozdíl od síťové technologie Ethernet nebo rozhraní USB se tedy jedná o zcela bezkolizní fyzickou vrstvu. V oblasti osobních počítačů se RS-232 již téměř nepoužívá, protože bylo nahrazeno výkonnějším USB. Nicméně v průmyslu je tento standard, především jeho modifikace – standardy RS-422 a RS-485, velice rozšířen.

SPI – Angl. Serial Peripheral Interface, je sériové periferní rozhraní, které se používá pro komunikaci mezi řídícími mikroprocesory a ostatními integrovanými obvody (EEPROM, A/D převodníky, displeje atd.) po společné sběrnici. Adresace se provádí pomocí zvláštních vodičů, které při logické nule aktivují příjem a vysílání zvoleného zařízení.

USART – Synchronní / asynchronní sériové rozhraní (angl. Universal Synchronous / Asynchronous Receiver and Transmitter). Lze jej nastavit buď pro asynchronní režim (SCI – např. pro linky RS232 resp. RS485), anebo pro synchronní režim (SPI).

USB – Angl. Universal Serial Bus, je univerzální sériová sběrnice, která nahrazuje dříve používané způsoby připojení periferií k počítači (sériový a paralelní port, PS/2, Gameport apod.). Nabízí moderní způsob propojení počítače s periferiemi jako jsou tiskárny, myši, klávesnice, fotoaparáty, modemy, paměťová média atd. Viz kapitola 3.2.

ZigBee – ZigBee je bezdrátová komunikační technologie vystavěná na standardu IEEE 802.15.4. Podobně jako Bluetooth je určena pro spojení nízkovýkonových zařízení v sítích PAN na malé vzdálenosti do 75 metrů. Díky použití multiskokového ad-hoc směrování umožňuje komunikaci i na větší vzdálenosti bez přímé radiové viditelnosti jednotlivých zařízení. Je určena hlavně pro aplikace v průmyslu a senzorových sítích. Pracuje v bezlicenčních pásmech (generální povolení) přibližně 868 MHz, 902–928 MHz a 2,4 GHz. Přenosová rychlost činí 20, 40, 250 kbit/s.

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

Přílohy

1 Úvod

Cílem této diplomové práce je vyzkoušení a ověření funkcionality volně dostupné a šiřitelné knihovny LibUSB pro obousměrnou komunikaci s libovolným mikrokontrolérem PIC řady 18 přes USB rozhraní.

Seznámíme se zde nejdříve s vlastnostmi mikrokontroléru PIC řady 18 a probereme, jaké možnosti a vybavení nabízí dnešní mikrokontroléry a jaké prostředky můžeme použít při oživování mikrokontroléru. Potom si povíme něco bližšího o USB rozhraní, jaké verze nabízí, jak vypadá topologie rozhraní a jakým způsobem je komunikační systém rozdělen na vrstvy. Popíšeme si knihovnu LibUSB pro Linux, jakým způsobem se používá, její základní funkce, jak identifikuje zařízení atd.

Ukážeme si základní zapojení mikrokontroléru PIC18F26J50 a plynule přejdeme na návrh programového vybavení v jazyce C pro mikrokontrolér tzv. firmware. Nejprve si ukážeme, jak vytvořit jednoduchý program, jak jej přemístit na mikrokontrolér. Potom popíši samotnou demonstrační aplikaci, která umožňuje obousměrnou komunikaci mezi počítačem a mikrokontrolérem přes USB rozhraní způsobem dotaz-odpověď (synchronně) a asynchronně. Seznámím Vás s návrhem odpovídajícího programového řešení v jazyce C++ s použitím knihovny LibUSB, aby byly splněny stejné požadavky, jako pro program mikrokontroléru. Na závěr si ukážeme výsledky testování stability, spolehlivosti, rychlosti, odezvy i komunikaci s více mikrokontroléry najednou.

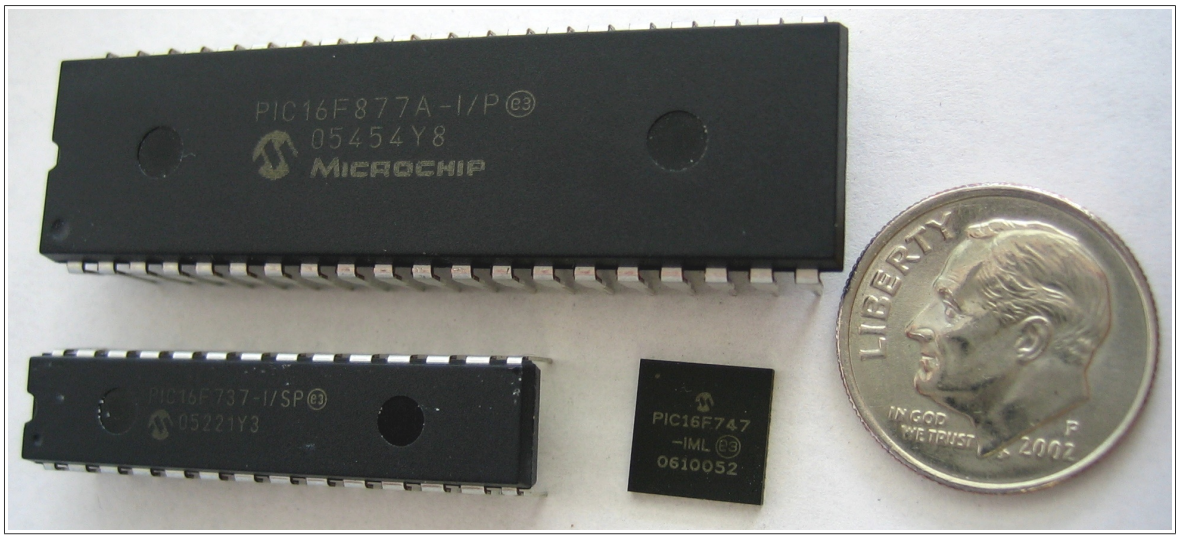
2 Mikrokontrolér PIC

2.1 Obecný popis mikrokontroléru

Mikrokontrolér PIC je monolitický integrovaný obvod obsahující kompletní mikropočítač. Na jediném čipu obsahuje mikroprocesor (CPU), paměť, programovatelné vstupně-výstupní rozhraní a další periferní obvody (viz obrázek 1). V mnoha zapojeních tedy nepotřebuje žádné další podpůrné obvody ke své činnosti. Je založen na harvardské architektuře, má tedy oddělenou paměť programu a dat. Obsahuje malé množství strojových instrukcí pevné délky (RISC) a vykonávání většiny instrukcí trvá jeden cyklus (4 hodinové takty). Dělí se podle šířky datového slova (8, 16 a 32 bitů), podle šířky programového slova (10, 12, 14 bitů, …) a podle typu paměti (OTP, EEPROM, FLASH) na následující řady:

- 8bitové mikrokontroléry: PIC10, PIC12, PIC14, PIC16, PIC17, PIC18
- 16bitové mikrokontroléry: PIC24
- 16bitové digitální signálové mikrokontroléry: dsPIC30, dsPIC33F
- 32bitové mikrokontroléry: PIC32 (MIPS)

Vyznačuje se velkou spolehlivostí a kompaktností a používá se především pro jednoúčelové aplikace jako je řízení, regulace apod. Více informací lze najít na internetových stránkách [1, 2].



Obrázek 1: PIC mikrokontroléry – převzato z [1]

2.2 Mikrokontrolér PIC18F

Mikrokontrolér PIC řady 18 nabízí největší výkon ze skupiny 8bitových jednočipových mikropočítačů, takže může obsahovat i vyspělejší periferie jako CAN, USB, Ethernet, LCD atd. Všechny dnes dostupné mikrokontroléry PIC řady 18 jsou vyráběny s interní pamětí typu FLASH, proto mají v názvu písmeno „F“ (déle tedy budeme používat název PIC18F nebo jen PIC18). Pokud název obsahuje písmena „LF“, pak PIC18 podporuje nižší napájecí napětí.

PIC18 se vyznačuje následujícími vlastnostmi:

- pouzdro má 18–100 vývodů
- výkon až 16 MIPS
- 83 instrukcí s délkou 16 bitů, které jsou optimalizovány pro jazyk C
- dokáže adresovat až 2 MB programové paměti
- až 4 KB RAM, až 128 KB interní programové paměti
- hardwarový zásobník – 32 úrovní
- max. pracovní frekvence začíná na 32 MHz, končí na 64 MHz, v drtivé většině 40 MHz
- pracovní teplota začíná na –40 °C, končí okolo 100 °C
- mnoho mikrokontrolérů PIC18 zahrnuje technologii nanoWatt XLP pro zařízení vyžadující velmi malý odběr elektrické energie

Podle označení se pak mikrokontroléry PIC18 dělí na tři skupiny resp. řady:

- tradiční řada – označení PIC18Fxxxx (např. PIC18F4550)
  - vysoce odolná FLASH paměť – až 100 000 přepisů, trvanlivost dat až 40 let
  - interní datová paměť EEPROM – až jeden milion přepisů, trvanlivost dat až 40 let
  - výkon pouze 10 MIPS (40 MHz) při napájení 5 V
  - napájecí napětí 2,0 – 5,5 V
- řada „J“ – označení PIC18FxxJxx (např. PIC18F46J50)
  - aplikace požadující co nejnižší cenu, ale nabízí široké spektrum funkcí
  - méně odolná FLASH paměť – až 10 000 přepisů, trvanlivost dat až 20 let
  - neobsahuje datovou paměť EEPROM – emulována interní programovou pamětí
  - střední výkon 10 – 12 MIPS (40 – 48 MHz) při napájení 3 V
  - napájecí napětí 2,0 – 3,6 V, některé vstupně-výstupní piny jsou tolerantní k napětí 5 V
  - pouzdra mají 28 – 100 vývodů
- řada „K“ – označení PIC18FxxKxx (např. PIC18F46K80)
  - méně odolná FLASH paměť – až 10 000 přepisů, trvanlivost dat až 40 let
  - interní datová paměť EEPROM – až 100 000 přepisů, trvanlivost dat až 40 let
  - největší výkon až 16 MIPS (64 MHz)
  - napájecí napětí 1,8 – 5,5 V
  - pouzdra mají 20 – 80 vývodů

PIC18 – řada „J“

Řada „J“ obsahuje přes 70 druhů mikrokontrolérů, které se dále dělí do podskupin podle toho, k jakému účelu jsou vhodné (viz tabulka 1).

Zaměření	Příklad typu mikrokontroléru	Specifika	Velikost paměti [KB]	Počet pinů	Výkon [MIPS]
Všeobecné použití	PIC18F45J10	–	16 – 32	28 – 44	10
Všeobecné použití	PIC18F87J10	–	32 – 128	64 – 80	10
Všeobecné použití	PIC18F46J11	XLP	16 – 64	28 – 44	12
Všeobecné použití	PIC18F87J11	–	8 – 128	64 – 80	10/12
USB	PIC18F46J50	USB, XLP	16 – 64	28 – 44	12
USB	PIC18F87J50	USB	32 – 128	64 – 80	12
Ethernet	PIC18F97J60	10 Mbit/s MAC/PHY	64 – 128	64 – 100	10
LCD	PIC18F85J90	LCD	8 – 32	64 – 80	10
LCD	PIC18F87J9x	LCD	64 – 128	64 – 80	12

Tabulka 1: Rozdělení mikrokontrolérů PIC18F do skupin

2.2.1 Vývojové nástroje

Firma Microchip nabízí také širokou škálu softwarových a hardwarových vývojových nástrojů.

Vývojové prostředí

Mezi softwarové nástroje patří především vývojové prostředí MPLAB IDE, které slouží k psaní programu v jazyce C, nebo v jazyce symbolických adres (Assembleru). V době, kdy jsem začal pracovat na této diplomové práci (červenec 2011), byly dostupné dvě verze. MPLAB IDE 8, která funguje pouze jako 32bitová aplikace v operačním systému Microsoft Windows a betaverze MPLAB X IDE, která pracuje v operačních systémech Microsoft Windows (x86/x64), Linux (32/64 bitů), Mac OS X 10.X. Zaznamenal jsem rychlý a intenzivní vývoj aplikace MPLAB X IDE v průběhu několika měsíců, kdy jsem vytvářel program pro mikrokontrolér (firmware). Pracoval jsem s několika betaverzemi a na konci roku 2011 vyšla verze 1.0. Nyní, v době psaní této diplomové práce, už je dostupná verze MPLAB X IDE v1.10, přičemž verze MPLAB IDE 8 se téměř nezměnila, takže se dá usoudit, že MPLAB X IDE v budoucnu úplně nahradí verzi MPLAB IDE 8.

Kompilátor

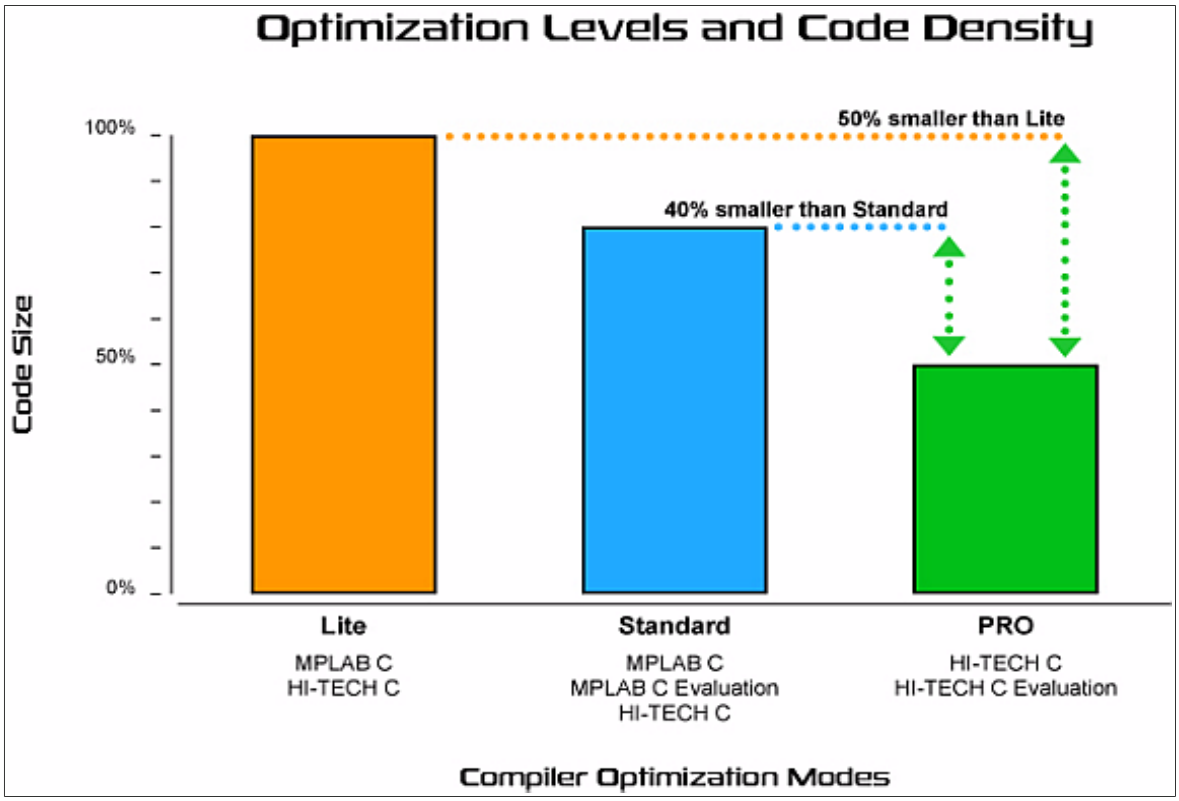
Nedílnou součástí je i kompilátor jazyka C popř. Assembleru. Microchip nabízí dvě rodiny, které se dále dělí do skupin podle toho, pro jaké řady mikrokontrolérů jsou určeny (viz tabulka 2) a podle licenčních podmínek firmy (viz tabulka 3). Liší se také podle toho, jak dokáží optimalizovat výsledný binární kód (viz obrázek 2).

MPLAB C	HI-TECH C
MPLAB C Compiler for PIC18 MCUs	PIC10/12/16 MCUs
MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs	PIC18 MCUs
MPLAB C Compiler for dsPIC DSCs	HI-TECH C Enterprise Edition
MPLAB C Compiler for PIC24 MCUs	
MPLAB C for PIC32 MCUs	

Tabulka 2: Rozdělení kompilátorů Microchip podle řady mikrokontroléru – převzato z [3]

	PRO	Standard	Lite	Evaluation
PIC10/12/16	HI-TECH C	HI-TECH C	HI-TECH C	HI-TECH C
PIC18	HI-TECH C	MPLAB C HI-TECH C	MPLAB C HI-TECH C	MPLAB C HI-TECH C
PIC24 a dsPIC		MPLAB C	MPLAB C	MPLAB C
PIC32		MPLAB C	MPLAB C	MPLAB C

Tabulka 3: Rozdělení kompilátorů Microchip podle licenčních podmínek – převzato z [3]



Obrázek 2: Úrovně optimalizace a hustota kódu kompilátorů Microchip – převzato z [3]

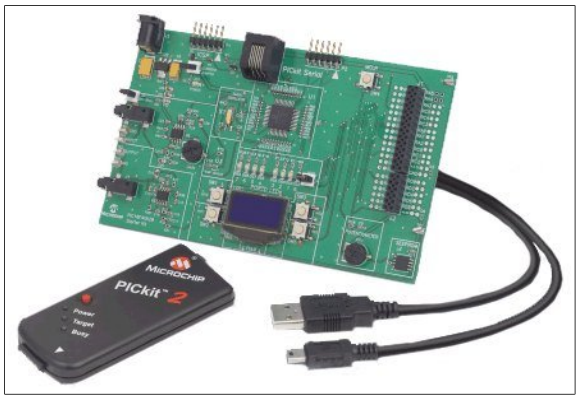
Programátor, debugger a vývojové desky

Další nedílnou součástí vývoje nějakého zařízení je programátor mikrokontroléru (angl. programmer) popř. ladící zařízení (angl. debugger). Microchip nabízí několik programátorů lišících se počtem podporovaných mikrokontrolérů, způsobem, jakým mohou spolupracovat s vývojovým prostředím MPLAB atd. Od toho se odvíjí i cena. Většinu nabízených programátorů lze připojit do obvodu, takže lze mikrokontrolér programovat přímo v obvodu. Tento systém se anglicky nazývá In-Circuit Debugger System. Na obrázcích 3, 4 a 5 jsou příklady některých programátorů. Ve svém sortimentu má firma Microchip i velké množství vývojových desek (angl. demo boards), které se liší množstvím periferií, použitým mikrokontrolérem, který je požadován výslednou aplikací atd. (viz obrázky 6 a 7).

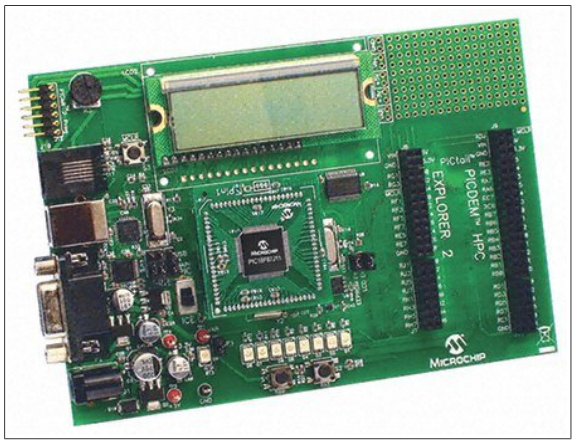
Obrázek 3: Programátor PICkit 3

Obrázek 4: Programátor MPLAB ICD 3

Obrázek 5: Programátor MPLAB REAL ICE



Obrázek 6: Vývojová deska PIC18F4XK20 Starter Kit



Obrázek 7: Vývojová deska PIC18 Explorer Board

2.3 Shrnutí

Pro splnění zadání jsem si vybral mikrokontrolér PIC18F26J50, který patří do rodiny mikrokontroléru PIC18F46J50. Obsahuje modul USB 2.0 – Full speed, má 64 KB interní programovou FLASH paměť, skoro 4 KB RAM a 28 vývodů, což by měly být více než dostačující parametry vzhledem k rozsahu diplomové práce.

Vybral jsem si volně dostupné vývojové prostředí MPLAB X IDE určené pro 32bitový operační systém Linux. Primárně jsem používal kompilátor MPLAB C18 v3.40 opět pro Linux. Z finančních důvodů jsem nepoužil žádný z programátorů firmy Microchip, ale zapůjčil jsem si USB programátor ASIX PRESTO od mého vedoucího diplomové práce.

3 Knihovna LibUSB

3.1 Základní popis

LibUSB je knihovna, která umožňuje komunikaci mezi USB zařízeními a uživatelskou aplikací. Její zdrojový kód je (v souladu s LGPL) volně dostupný a lze jej libovolně měnit a přizpůsobovat. Je napsána v jazyce C a podporuje několik operačních systémů. Programátoři používající knihovnu LibUSB by měli mít základní znalost ovládání USB zařízení z programátorského hlediska. Úplné minimum se pokusím nastínit v další kapitole 3.2 a potom podrobněji popíši knihovnu LibUSB.

3.2 Komunikační systém USB 2.0

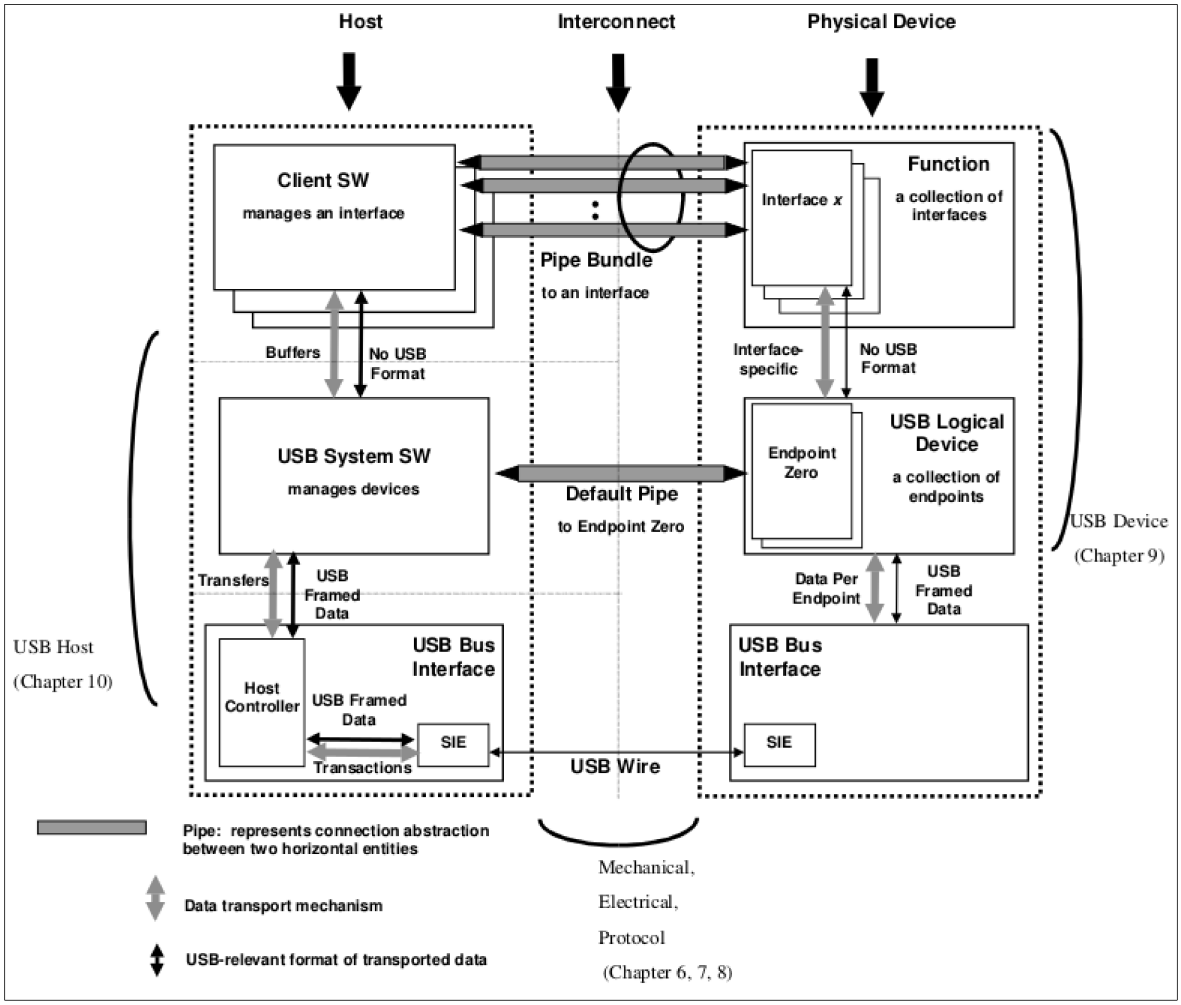
Pro větší přehlednost a usnadnění práce různým výrobcům je celý komunikační systém rozdělen do několika pohledů resp. vrstev. Obrázek 8 znázorňuje jednotlivé vrstvy a jejich interakce. Komunikace mezi kořenovým uzlem (angl. Host) a koncovým fyzickým zařízením (angl. Physical Device) vyžaduje interakci mezi několika vrstvami a entitami:

- Vrstva rozhraní USB sběrnice (angl. USB Bus Interface layer): poskytuje fyzickou, signálovou či rámcovou konektivitu mezi kořenovým uzlem a koncovým zařízením
- Vrstva USB zařízení (angl. USB Device layer): slouží pro výměnu informací mezi softwarem USB systému, který provádí obecné USB operace s USB zařízením
- Funkční vrstva (angl. Function layer): poskytuje další schopnosti kořenovému uzlu skrz odpovídající vrstvu klientského softwaru

Z hlediska implementace rozlišujeme 4 základní oblasti:

- Fyzické USB zařízení (angl. USB Physical Device): je to kus hardwaru na konci USB kabelu, který plní nějakou funkci
- Klientský software (angl. Client Software): je spuštěn na kořenovém uzlu (PC), je většinou součástí operačního systému nebo je dodáván spolu s USB zařízením
- Software USB systému (angl. USB System Software): podporuje USB v daném operačním systému, je většinou součástí operačního systému a je nezávislý na konkrétním USB zařízení nebo klientském softwaru
- Kontrolér kořenového uzlu (angl. USB Host Controller nebo také Host Side Bus Interface): hardware a software, který umožňuje připojení koncového zařízení ke kořenovému uzlu

Vrstva USB zařízení a funkční vrstva mají své vlastní logické/virtuální komunikační proudy se svými protilehlými vrstvami (na opačném zařízení), ale obě ve skutečnosti používají vrstvu rozhraní USB sběrnice k přenosu dat. Obrázek 8 také nabízí detailní pohled na implementaci komunikace mezi kořenovým uzlem a koncovým zařízením. Můžeme z něj vyčíst i odkazy na kapitoly v literatuře [4] (Chapter 6 – 10), kde lze najít více informací.



Obrázek 8: Detailní pohled na implementaci USB komunikace – převzato z [4]

USB poskytuje komunikační službu mezi softwarem na kořenovém uzlu a jeho USB funkce. Funkce mohou mít různé požadavky na datový tok pro různé interakce mezi klientem a funkcí. Možností oddělení datových toků dochází k lepšímu využití sběrnice. Každý datový tok je ohraničen koncovým bodem (angl. endpoint) a využívá tak sběrnici pro komunikaci pouze ve vymezeném čase. Koncové body jsou používány k identifikaci stavu každého datového toku.

Logické zařízení USB se jeví jako kolekce koncových bodů, které dohromady tvoří rozhraní (ang. interface). Tato rozhraní jsou pak pohledy na funkci. Software USB systému řídí zařízení pomocí výchozího řídícího propojení (angl. Default Control Pipe). Klientský software pak řídí rozhraní pomocí balíků propojení (angl. pipe bundles). Navíc také požaduje data, která jsou přenášena přes USB mezi vyrovnávací pamětí kořenového uzlu a koncovém bodu USB zařízení. Kontrolér kořenového uzlu (nebo USB zařízení, záleží na směru přenosu dat) zabalí data do tzv. paketu, který je přenášen přes USB, a rozhoduje o tom, kdy je sběrnice obsazena pro přenos paketu.

3.2.1 Koncové body zařízení (angl. Endpoints)

V předchozím odstavci jsem se zmínil o koncových bodech zařízení. Rád bych je blíže popsal, protože jsou velice užitečné (a nezbytné) při komunikaci přes USB.

Koncový bod je jedinečná identifikovatelná část USB zařízení, která slouží jako začátek nebo konec (podle směru přenosu dat) komunikačního toku mezi kořenovým uzlem a koncovým zařízením. Každé logické USB zařízení je tvořeno kolekcí na sobě nezávislých koncových bodů a má jedinečnou adresu přiřazenou systémem v době připojení zařízení. Každému koncovému bodu je v době návrhu přiřazen jedinečný identifikátor (v rámci jednoho zařízení), který se nazývá číslo koncového bodu (angl. endpoint number). Každý koncový bod má navíc přiřazen směr přenosu dat. Kombinací adresy zařízení, čísla koncového bodu a směru přenosu dat získáváme jedinečný ukazatel na koncový bod. Koncový bod je tedy jednosměrné spojení, které může být buď odchozí (od kořenového uzlu ke koncovému zařízení), nebo příchozí (od koncového zařízení ke kořenovému uzlu).

Všechna USB zařízení musí implementovat výchozí řídící funkci, která používá oba, odchozí i příchozí, koncové body s číslem koncového bodu 0 (nula). Software USB systému používá tuto funkci k inicializaci a k všeobecné manipulaci logického zařízení (např. konfigurace logického zařízení). Výchozí řídící propojení umožňuje přístup ke konfiguračním informacím koncového zařízení, mění stav zařízení atd. Koncové body 0 jsou vždy dostupné, když je koncové zařízení připojeno, napájeno a obdrželo požadavek na výchozí nastavení sběrnice.

Funkce mohou mít přiřazeny další koncové body. LOW-SPEED funkce mohou mít navíc pouze dva koncové body a FULL-SPEED zařízení mohou mít takový počet koncových bodů, který je definován protokolem (např. maximálně 15 odchozích a 15 příchozích). Tyto přídavné koncové body nemohou být používány dokud není zařízení nakonfigurováno.

3.3 Verze LibUSB

V současné době existují dvě základní verze:

- libusb-1.0: aktuální verze, která je aktivně vyvíjena
- libusb-0.1: originální verze, která není doporučována a již není vyvíjena několik let

Navíc je dostupná verze pro Windows jako samostatný projekt:

- libusb-win32: samostatný projekt vycházející z verze libusb-0.1

3.3.1 libusb-1.0

Projekt přebral Daniel Drake v lednu 2008, který pracoval na libusb-1.0 API a implementaci, přidává několik vlastností, které chybí v libusb-0.1 API. V současné době je vývoj veden Danielem

Drakem a Peterem Stugem. libusb-1.0 je doporučovanou verzí, která podporuje operační systémy: Linux, Mac OS X, Windows, OpenBSD a FreeBSD (obsahuje vlastní implementaci libusb-0.1 a libusb-1.0 API v systému libc). Knihovna je kompletní a nová vydání by měla obsahovat pouze opravy chyb. Vnitřní vylepšení a nové funkce budou s největší pravděpodobností implementovány ve verzi libusb-1.1. Více informací lze najít v literatuře [5].

3.3.2 libusb-0.1

Johannes Erdfelt založil projekt LibUSB a vedl vývoj do roku 2007. Do té doby se verze libusb-0.1 stabilizovala a byla použita ve velkém množství projektů. Poslední verze libusb-0.1.12 podporuje operační systémy Linux, FreeBSD, NetBSD, OpenBSD, Darwin, Mac OS X (a Windows – projekt libusb-win32). Více informací lze najít v kapitolách 3.4 a 3.5 a v literatuře [5].

3.4 Základní vlastnosti knihovny libusb-1.0

- podporuje všechny typy přenosu dat (angl. control/bulk/interrupt/isochronous)
- má dvě rozhraní pro přenos dat
  - synchronní (jednoduchý na použití)
  - asynchronní (složitější, ale efektivnější)
- spolehlivý z pohledu vláken (ani pro asynchronní rozhraní není zapotřebí vláken)
- malé množství kódu při použití API
- kompatibilní s libusb-0.1 skrz překladovou vrstvu libusb-compat-0.1

3.4.1 Synchronní a asynchronní rozhraní

LibUSB nabízí dvě navzájem oddělaná rozhraní pro vstup a výstup zařízení, která mohou být aplikací používána současně. Z logického hlediska se přenos dat přes USB děje ve dvou krocích. Např. pokud jsou data čtena z koncového bodu:

1. Požadavek pro čtení dat je zaslán koncovému zařízení.
2. Data jsou po nějakém čase přijata kořenovým uzlem.

nebo v případě zápisu dat na koncový bod:

1. Data jsou zaslána koncovému zařízení.
2. Po nějakém čase kořenový uzel obdrží potvrzení od koncového bodu, že data byla úspěšně přenesena.

Mezi oběma body může být předem nedefinovatelné zpoždění. Představme si například aplikaci s jedním tlačítkem. Pokud uživatel stiskne tlačítko, požadavek na čtení dat je zaslán koncovému zařízení a aplikace čeká, dokud nedorazí odpověď, což může být čistě teoreticky i několik hodin. Takovýto scénář by se mohl stát při použití synchronního rozhraní. V případě asynchronního přenosu by aplikace poslala požadavek koncovému zařízení a pokračovala by ve vykonávání programu. Čtení odpovědi se provede až v některém následujícím kroku. Základní rozdíl mezi

oběma přístupy tedy je, že synchronní rozhraní kombinuje oba výše popsané kroky v jednom volání funkce, ale asynchronní rozhraní tyto kroky odděluje.

Synchronní rozhraní

Jak již bylo popsáno výše synchronní rozhraní provede přenos dat pomocí jednoho volání funkce, po kterém je přenos ukončen a může být zpracován výsledek. Stejný princip byl používán i v předchozí verzi libusb-0.1, která obsahovala pouze synchronní rozhraní. Příklad zdrojového kódu 1 ukazuje, jakým způsobem lze knihovnu použít pro synchronní přenos dat. Hlavní výhodou tohoto kódu je jeho jednoduchost. Toto rozhraní však má svá omezení. Aplikace bude uspána ve funkci `libusb_bulk_transfer()` dokud nebude transakce ukončena. Pokud bude např. stlačení tlačítka trvat 3 hodiny, aplikace bude celou tuto dobu spát. Vykonávání programu bude svázáno uvnitř knihovny, takže vlákno bude po celou dobu nepoužitelné. Dalším problémem svázání vlákna s transakcí je znemožnění provádění vstupně-výstupních operací přes více koncových bodů nebo s více zařízeními najednou, pod však není použito jedno vlákno pro každou transakci. V neposlední řadě je zde nemožnost zrušit přenos dat poté, co je odeslán požadavek.

```
unsigned char data[4];
int actual_length;
int r = libusb_bulk_transfer(handle, EP_IN, data, sizeof(data),
        &actual_length, 0);
if (r == 0 && actual_length == sizeof(data))
{
    // results of the transaction can now be found in the data buffer
    // parse them here and report button press
}
else
{
    error();
}
```

Zdrojový kód 1: Synchronní rozhraní – převzato z [6]

Asynchronní rozhraní

Asynchronní vstupně-výstupní rozhraní je nejdůležitější charakteristikou knihovny libusb-1.0. I když je více složitější, řeší všechny problémy popsány výše. Místo blokujících funkcí nabízí asynchronní rozhraní neblokující funkce, které po zavolání zahájí přenos dat a okamžitě skončí. Jeden z parametrů takové funkce je ukazatel na funkci zpětného volání (angl. callback function), kterou knihovna LibUSB zavolá po ukončení transakce spolu s jejím výsledkem. Přenosy dat, které byly zahájeny přes neblokující funkce mohou být zrušeny zavoláním oddělené funkce. Toto rozhraní umožňuje souběžně provádět vstupně-výstupní operace s více koncovými body i s více zařízeními bez nutnosti použití vláken.

Tato nová flexibilita však přináší nové komplikace. Z důvodu úspory velikosti kódu knihovny, LibUSB nevytváří vlákna. Funguje pouze tak, že klientská aplikace provádí volání knihovny ze své hlavní smyčky a zjišťuje tak, jestli jsou události připraveny k obsluze, nebo musí být použit jiný způsob, jak umožnit knihovně převzít kontrolu a obsloužit tak prováděné transakce. Navíc musí být LibUSB pravidelně volána v určitém definovaném čase, aby bylo zajištěno správného obsloužení vypršení času transakce. Dalším problémem je složitější obsluha paměti. Vyrovnávací paměť (angl. buffer) nemůže být využívána, dokud nějaká jiná funkce, používající vyrovnávací paměť, neukončí transakci skrz svou funkci zpětného volání. Ztrácí se také jistá linearita kódu, protože zahájení přenosu se provádí v jedné funkci a výsledek je zpracován v jiné funkci.

3.5 Práce s knihovnou libusb-1.0

3.5.1 Inicializace a uvolnění knihovny

Před použitím jakékoliv funkcionality knihovny je nutné LibUSB inicializovat. K tomu slouží funkce

```
int libusb_init(libusb_context **context).
```

Pokud už nechceme s knihovnou dále pracovat, měli bychom ji uvolnit funkcí

```
void libusb_exit(struct libusb_context *ctx).
```

V obou funkcích je použit parametr typu `libusb_context`, což je struktura reprezentující relaci nebo-li sezení (angl. session). Tento koncept relací umožňuje programu používat dvě knihovny (nebo dynamicky načíst dva moduly), které používají LibUSB nezávisle na sobě. Tím zabráníme jednotlivým uživatelům v jejich vzájemném rušení. Pokud např. zavoláme funkci `libusb_exit()` s jedním kontextem, nedojde k uvolnění zdrojů, které využívá jiný uživatel ve svém kontextu. Relace jsou vytvářeny funkcí `libusb_init()` a rušeny funkcí `libusb_exit()`. Pokud aplikace používá pouze jednu knihovnu LibUSB, není nutné se zabývat kontexty. Místo toho je možné poslat NULL místo parametru `libusb_context`, čímž bude použit výchozí kontext.

3.5.2 Výpis seznamu zařízení a obsluha zařízení

V této části si popíšeme, jak získat seznam USB koncových zařízení připojených k operačnímu systému, jak vybrat požadované zařízení a jak ho otevřít a zavřít. K tomu je zapotřebí znát další dvě struktury obsažené v LibUSB:

- `libusb_device`: reprezentuje USB koncové zařízení, které je nebo bylo nedávno připojené k operačnímu systému. Pomocí této struktury je možno získat jisté omezené informace o koncovém zařízení (např. data deskriptoru – viz dále), nezískáme však aktuální stav zařízení (připojeno/odpojeno, práva pro používání zařízení, zda je zařízení používáno jiným programem atd.)
- `libusb_device_handle`: reprezentuje ovladač USB koncového zařízení. Díky této struktuře můžeme otevřít USB koncové zařízení a provádět na něm vstupně-výstupní operace.

K získání seznamu všech aktuálně připojených USB koncových zařízení k operačnímu systému nám slouží funkce:

```
ssize_t libusb_get_device_list(
      libusb_context *ctx, libusb_device ***list).
```

Tento seznam musí být uvolněn, když už není zapotřebí, k čemuž slouží funkce:

```
void libusb_free_device_list(
      libusb_device **list, int unref_devices).
```

Samotné USB koncové zařízení pak otevřeme resp. zavřeme pomocí funkcí:

```
int libusb_open(libusb_device *dev, libusb_device_handle **handle)
```

resp.

```
void libusb_close(libusb_device_handle *dev_handle).
```

Celý proces nalezení požadovaného USB koncového zařízení může vypadat takto:

1. Získej seznam všech USB koncových zařízení – `libusb_get_device_list()`
2. Vyber požadované zařízení podle ID výrobce, ID produktu popř. dalších parametrů (v našem případě podle sériového čísla) a otevři ho – `libusb_open()`
3. Zruš ukazatele na všechna USB koncová zařízení – `libusb_free_device_list()`
4. Uvolni seznam všech USB koncových zařízení – `libusb_free_device_list()`

Důležité je dodržení tohoto pořadí – koncové zařízení musí být otevřeno dřív než je na něj zrušena reference. Příklad zdrojového kódu 2 pak ukazuje, jak může vypadat nalezení USB koncového zařízení a jeho otevření v jazyce C. Tato sekvence volání funkcí bude vyhovovat téměř všem scénářům, přičemž není zapotřebí hlubšího porozumění problému obsluhy sdílených zdrojů.

```
// discover devices
libusb_device **list;
libusb_device *found = NULL;
ssize_t cnt = libusb_get_device_list(NULL, &list);
ssize_t i = 0;
int err = 0;
if (cnt < 0)
    error();

for (i = 0; i < cnt; i++)
{
    libusb_device *device = list[i];
    if (is_interesting(device))
    {
        found = device;
        break;
    }
}

if (found)
{
    libusb_device_handle *handle;
    err = libusb_open(found, &handle);
    if (err)
        error();
    // etc
}

libusb_free_device_list(list, 1);
```

Zdrojový kód 2: Nalezení USB koncového zařízení – převzato z [6]

3.5.3 Struktury USB deskriptorů a jejich čtení

Každé USB zařízení musí umět sdělit svému okolí, jaké má atributy. To se děje pomocí tzv. deskriptorů (angl. descriptors), což jsou datové struktury s přesně definovaným formátem. Každý deskriptor začíná polem délky jednoho bytu obsahující celkový počet bytů daného deskriptoru. Následuje pole délky jednoho bytu obsahující typ deskriptoru. Další pole se pak liší podle typu deskriptoru. LibUSB obsahuje 4 datové struktury reprezentující standardní USB deskriptory:

- deskriptor zařízení (angl. device descriptor)
  - `struct libusb_device_descriptor`
- deskriptor koncového bodu (angl. endpoint descriptor)
  - `struct libusb_endpoint_descriptor`
- deskriptor rozhraní (angl. interface descriptor)
  - `struct libusb_interface_descriptor`
- deskriptor konfigurace (angl. configuration descriptor)
  - `struct libusb_config_descriptor`

Další deskriptory (např. textový – angl. string descriptor) jsou implementovány jiným způsobem (např. polem znaků).

Datové struktury deskriptor zařízení, konfigurace a textový deskriptor mohou být získány jim vyhrazenými funkcemi:

```
int libusb_get_device_descriptor(
      libusb_device *dev, struct libusb_device_descriptor *desc),
int libusb_get_config_descriptor(
      libusb_device *dev, uint8_t config_index,
      struct libusb_config_descriptor **config),
int libusb_get_string_descriptor_ascii(
      libusb_device_handle *dev, uint8_t desc_index,
      unsigned char *data, int length).
```

Ostatní deskriptory jsou získatelné pomocí funkce:

```
static int libusb_get_descriptor(
      libusb_device_handle *dev, uint8_t desc_type,
      uint8_t desc_index, unsigned char *data, int length).
```

3.5.4 Synchronní komunikace

Synchronní komunikace resp. rozhraní bylo již popsáno v kapitole 3.4.1. Použití je relativně jednoduché. LibUSB obsahuje pouze tři funkce lišící se typem přenosu:

- kontrolní přenosy – `libusb_control_transfer`
- přenos většího objemu dat – `libusb_bulk_transfer`
- přenos přerušení – `libusb_interrupt_transfer`

3.5.5 Asynchronní komunikace

Asynchronní rozhraní, které nabízí knihovna libusb-1.0, je neblokující, protože na rozdíl od synchronního rozhraní od sebe odděluje odeslání požadavku na přenos dat a samotný přenos dat včetně jeho dokončení. Tím je možno se vyhnout potenciálně dlouhému zpoždění během přenosu dat.

Z důvodu abstrakce, LibUSB obsahuje pro asynchronní komunikaci obecnou strukturu s názvem `libusb_transfer`, která zahrnuje všechny typy přenosu dat (angl. control, bulk, interrupt, isochronous).

Komunikace probíhá v následujících pěti krocích:

1. **Alokace**: alokuje se paměť pro strukturu přenosu `libusb_transfer`
2. **Naplnění**: struktura přenosu `libusb_transfer` se naplní informacemi o přenosu, který se má uskutečnit
3. **Odeslání**: požadavek na přenos dat se pošle knihovně
4. **Obsluha dokončeného přenosu dat**: zpracuje se výsledek přenosu uložený ve struktuře přenosu `libusb_transfer`
5. **Uvolnění**: dealokuje se paměť

Uživatel může navíc využít další dva alternativní scénáře:

1. **Znovu-odeslání**: požadavek na přenos dat se znovu pošle knihovně
2. **Zrušení**: zruší právě probíhající přenos dat

Alokace

Tento krok zahrnuje alokaci paměti pro strukturu přenosu dat `libusb_transfer`, která zatím obsahuje pouze předdefinovaná data. K tomu slouží následující funkce:

```
struct libusb_transfer* libusb_alloc_transfer(int iso_packets)
```

Naplnění

Struktura alokovaná v předchozím kroku je naplněna informacemi, jako typ zprávy, směr přenosu dat, vyrovnávací paměť, funkce zpětného volání (angl. callback function) atd. Podle typu přenosu můžeme použít jednu z následujících funkcí:

- `libusb_fill_control_transfer()`,
- `libusb_fill_bulk_transfer()`,
- `libusb_fill_interrupt_transfer()`.

Odeslání

Pokud máme naplněnou strukturu přenosu dat `libusb_transfer`, můžeme odeslat požadavek pro přenos dat pomocí následující neblokující funkce:

```
int libusb_submit_transfer (struct libusb_transfer * transfer)
```

Obsluha dokončeného přenosu dat

Po odeslání požadavku na přenos dat může nastat jedna z následujících událostí:

- přenos dat byl dokončen (např. nějaká data byla přenesena)
- pro přenos dat je nastaven časový limit, který vypršel dřív, než byla přenesena všechna data
- přenos dat selhal v důsledku nějaké chyby
- přenos dat byl zrušen

V každém případě je zavolána funkce zpětného volání typu:

```
void(* libusb_transfer_cb_fn)(struct libusb_transfer *transfer),
```

ve které může být rozhodnuto, jak dále postupovat podle výše uvedeného typu události. Této funkci je předán parametr struktury přenosu dat `libusb_transfer`, který po ukončení přenosu obsahuje informace, jako stav (úspěch nebo neúspěch včetně důvodu neúspěchu), počet přenesených bytů atd.

Uvolnění

Pokud je přenos dat ukončen (např. funkce zpětného volání byla zavolána) a není požadován další přenos dat, je doporučeno uvolnit alokovanou paměť pomocí následující funkce:

```
void libusb_free_transfer(struct libusb_transfer * transfer)
```

Znovu-odeslání

Díky oddělení kroků alokace, naplnění a odeslání je možno znovu odeslat požadavek na přenos dat, pro který máme naplněnou strukturu přenosu dat `libusb_transfer`. Není tedy nutné znovu alokovat paměť a plnit strukturu `libusb_transfer`, což je užitečné v případech, kdy je potřeba opětovného přenosu stejného typu dat (např. periodické čtení teploty).

Zrušení

Další výhodou asynchronního rozhraní je možnost přerušení právě probíhajícího přenosu dat. K tomu slouží neblokující funkce:

```
int libusb_cancel_transfer(struct libusb_transfer * transfer)
```

Po zrušení přenosu dat je opět zavolána funkce zpětného volání, ve které lze zkontrolovat, jestli byl přenos dat přerušen. V takovém případě mohla být část dat přenesena.

Obsluha událostí

Z důvodu úspory velikosti kódu knihovny, LibUSB nevytváří vlákna, což znamená, že pracuje pouze a jenom, když je volána z nadřazené aplikace. Model asynchronního přenosu dat však vyžaduje, aby knihovna prováděla práci v různém čase (např. tvorba výsledku vyžádaného přenosu dat a volání funkce zpětného volání). Z toho důvodu je nutné, aby aplikace volala knihovní funkci `libusb_handle_events()`, když je od LibUSB očekávána práce.

Dokumentace LibUSB [6] popisuje několik možností obsluhy událostí, resp. kdy přesně volat funkci `libusb_handle_events()` resp. `libusb_handle_events_timeout()`:

- periodicky v neblokujícím módu v krátkém intervalu z hlavní smyčky aplikace,
- opakovaně v blokujícím módu z vyhrazeného vlákna,
- integrovat LibUSB do hlavní smyčky aplikace za použití tzv. deskriptorů souboru (angl. file descriptors).

Já jsem použil modifikovanou poslední možnost. V ukázce zdrojového kódu 3 lze vidět, jak dochází k obsluze událostí. Nejprve se zavolá funkce `libusb_get_pollfds()`, která získá pole deskriptorů souboru. Po převedení tohoto pole na pole struktur `struct pollfd` se zavoláním následující funkce z knihovny `poll.h`:

```
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

zkontroluje, jestli na některém z deskriptorů došlo k nějaké aktivitě. Pokud ano, je nutné zavolat funkci knihovny LibUSB:

```
int libusb_handle_events_timeout(
      libusb_context * ctx, struct timeval * tv),
```

v neblokujícím módu, která obslouží čekající události. Více informací lze najít v kapitole 6.2.4 popř. v literatuře [6].

```
const libusb_pollfd **libusbFileDescriptors =
    libusb_get_pollfds(libusbContext);

nfds_t numberOfLibusbFileDescriptors = 0;
while (libusbFileDescriptors[numberOfLibusbFileDescriptors] != NULL)
    numberOfLibusbFileDescriptors++;

if (numberOfLibusbFileDescriptors > 0)
{
    struct pollfd pollFileDescriptors[numberOfLibusbFileDescriptors];
    for (int i = 0; i < (int)numberOfLibusbFileDescriptors; i++)
    {
        pollFileDescriptors[i].fd = libusbFileDescriptors[i]->fd;
        pollFileDescriptors[i].events =
            libusbFileDescriptors[i]->events;
    }

    // check if poll() indicated activity on libusb file descriptors
    if (poll(pollFileDescriptors, numberOfLibusbFileDescriptors, 0)>0)
        libusb_handle_events_timeout(libusbContext, 0);
}
```

Zdrojový kód 3: Obsluha událostí

3.5.6 Ošetření chybových stavů

Knihovní funkce typicky vrací 0 v případě úspěchu, nebo záporné číslo v případě neúspěchu. Tato negativní čísla se vztahují k výčtovému typu `LIBUSB_ERROR`, který obsahuje stručný popis chybových kódů.

3.5.7 Ladění aplikace

Za použití výchozího nastavení knihovna LibUSB nevypisuje ani nezaznamenává žádné ladící nebo chybové zprávy. Nabízí však funkci:

```
void libusb_set_debug(libusb_context * ctx, int level),
```

která nastavuje knihovnu LibUSB tak, aby vypisovala určité informace na standardní výstup nebo chybový výstup, což napomáhá při zjišťování problémů. Podle zadané úrovně (parametr `level`) se vypisují následující typy zpráv:

- 0: nejsou vypisovány žádné zprávy (výchozí nastavení)
- 1: chybové zprávy jsou vypisovány na standardní chybový výstup
- 2: varovné a chybové zprávy jsou vypisovány na standardní chybový výstup
- 3: informativní zprávy jsou vypisovány na standardní výstup a varovné a chybové zprávy jsou vypisovány na standardní chybový výstup

Vypisované zprávy nejsou nijak strukturované a neodpovídají návratovým hodnotám funkcí v případě chybových kódů. Nejsou ani lokalizované a podle tvůrců LibUSB ani nebudou. Proto se nedoporučuje zprávy nijak rozebírat ani zobrazovat uživateli. Místo toho by měly být interpretovány návratové chybové kódy, podle kterých by měl být uživatel informován o chybách.

3.6 Alternativní knihovny

Podle literatury [7] existuje několik projektů, které nabízí knihovny obalující libusb-0.1 (angl. libusb wrappers) a umožňují tak používat knihovnu LibUSB i z jiných programovacích jazyků než C/C++ (např. Python, Perl, Ruby, Java, FreePascal, C#.NET, Lua). Tím se rozšiřuje možnost použití knihovny LibUSB. Existuje také projekt OpenUSB, který je podle literatur [8, 9] odnoží knihovny libusb-1.0. Přináší některá rozšíření, jakými jsou např. podpora operačního systému Solaris (OpenSolaris, Solaris Nevada) a možnost připojení/odpojení USB zařízení za běhu aplikace.

3.7 Shrnutí

Pro tuto diplomovou práci jsem si vybral verzi knihovny libusb-1.0-0 (2:1.0.8-4), která je dostupná pro volně šiřitelnou distribuci Linuxu Ubuntu Oneiric (11.10). Podrobnější informace o použití knihovny LibUSB lze najít v její dokumentaci [6].

4 Zapojení mikrokontroléru

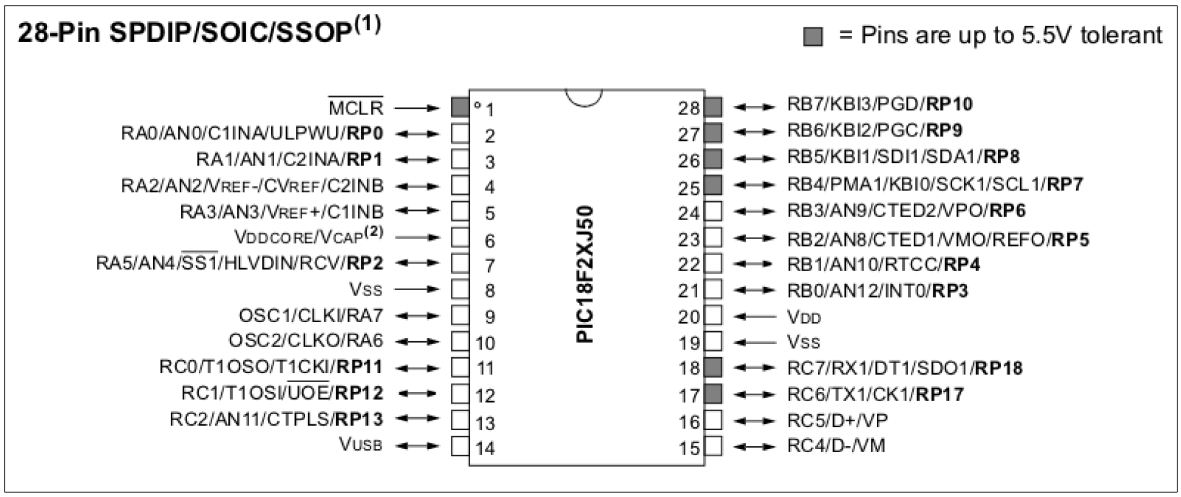
V této kapitole popíši jak pracovat s mikrokontrolérem PIC18F26J50, který patří do rodiny PIC18F46J50. Pokud nebude uvedeno jinak, budu psát právě o PIC18F26J50.

4.1 Popis mikrokontroléru PIC18F26J50

Tento mikrokontrolér má následující vlastnosti:

- 28 vývodů (z toho 16 programovatelných)
- 64 KB interní programová FLASH paměť
- 3776 bytů RAM
- modul USB 2.0
  - Full speed (12 Mbit/s) a Low speed (1,5 Mbit/s)
  - podporuje všechny typy přenosu dat (angl. control/bulk/interrupt/isochronous)
  - podporuje až 32 koncových bodů (angl. Endpoints) nebo 16 obousměrných
- velice přesný vnitřní oscilátor (typická tolerance ±0,15%) použitelný i pro USB, který může dodávat frekvenci až 48 MHz (31 kHz – 8 MHz, nebo až 48 MHz při použití PLL)
- 2 moduly MSSP podporující 3vodičové SPI (všechny 4 módy) a I^2C (master i slave módy)
- 2 moduly USART podporující RS-485, RS-232 a LIN/J2602

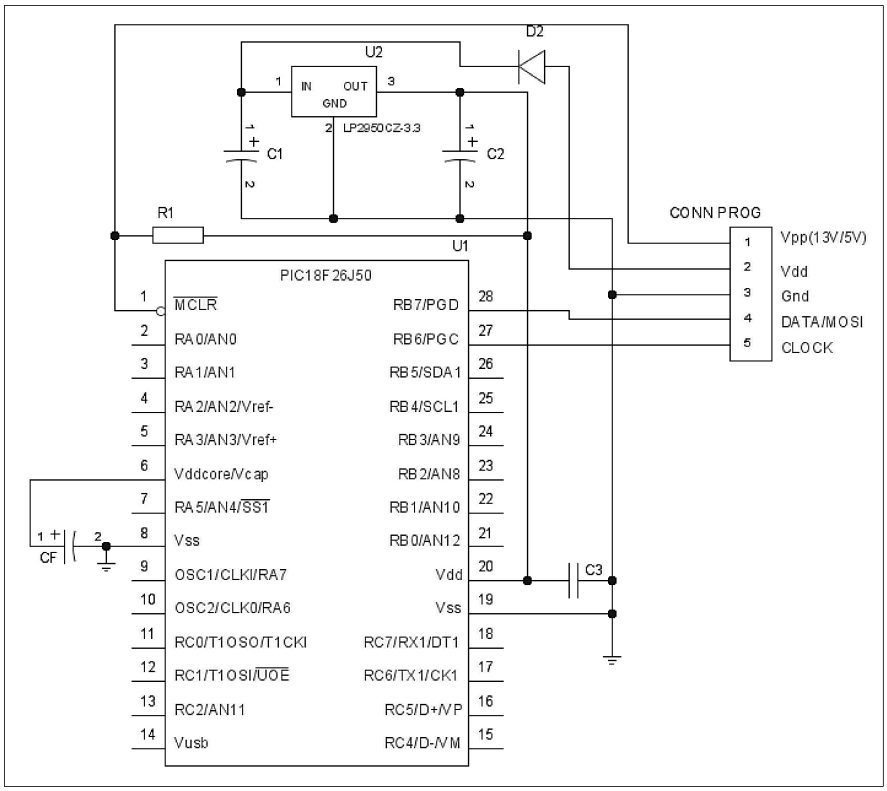
Obrázek 9 znázorňuje rozložení vývodů PIC18F26J50 – 28vývodové pouzdro SPDIP. Můžeme z něj mimo jiné vyčíst, že mikrokontrolér obsahuje 3 porty (A, B, C). U portu B můžeme využít všech 8 bitů, ale u portů A a C pouze 7 bitů (pouzdro neobsahuje vývody RA4 a RC3).



Obrázek 9: PIC18F26J50, 28vývodové pouzdro – převzato z [10]

4.2 Nejjednodušší zapojení mikrokontroléru PIC18F26J50

Pro oživení mikrokontroléru a jeho následné programování je nutné vytvořit minimální obvod zapojení, který je na obrázku 10. K mikrokontroléru je přivedeno napájecí napětí, v našem případě 3,3 V. To dodává regulátor napětí LP2950CZ-3.3 (U2), který je napájen přes diodu D2 přímo z programátoru (CONN PROG = konektor programátoru). Rezistor R1 nastavuje logickou jedničku na negovaném vstupu $\overline{\text{MCLR}}$ (angl. Master Clear). Tento vstup slouží ke dvěma účelům: resetování, nebo k programování a ladění mikrokontroléru. Kondenzátory C1 a C2 s kapacitou 1 µF slouží jako oddělovací kondenzátory pro regulátor napětí. Kondenzátor C3 slouží jako oddělovací kondenzátor mikrokontroléru a měl by být keramický s kapacitou 100 nF pro napětí minimálně 20 V. Kondenzátor CF slouží ke stabilizaci výstupního napětí vnitřního regulátoru napětí mikrokontroléru. Měl by mít kapacitu 10 µF a jeho maximální povolené napětí by mělo být minimálně 6,3 V u tantalového kondenzátoru, nebo minimálně 10 V u keramického. Všechny kondenzátory by měly být umístěny co nejblíž k součástce, ke které patří, vzdálenost nesmí přesáhnout 6 mm. Na vývod V_{DDCORE}/V_{CAP} nesmí být připojeno žádné napájecí napětí, jako tomu je na vývodu V_{DD}. Situace se liší v případě použití mikrokontroléru pro nízké napájecí napětí (řada PIC18LFXXJ50). Zde bych však odkázal čtenáře na literaturu [10] (strana 430), kde je tato specifická situace popsána. Stačí nám tedy pouze několik součástek, abychom zprovoznili mikrokontrolér resp. abychom ho připravili na programování.



Obrázek 10: Nejjednodušší zapojení mikrokontroléru PIC18F26J50

4.3 Schéma zapojení demonstrační aplikace

Pro demonstraci komunikace mezi počítačem a mikrokontrolérem přes USB rozhraní jsem navrhl zapojení na obrázku 11. Je to vlastně rozšíření základního zapojení popsaného v předchozí kapitole 4.2. Obsahuje navíc digitální teploměr (U3), 2 mikrospínače (S1, S2), 3 indikační svítivé diody (LED0, LED1, LED2), sedmisegmentový LED displej (D8), USB konektor a několik předřadných nebo omezujících rezistorů (R0 – R9).

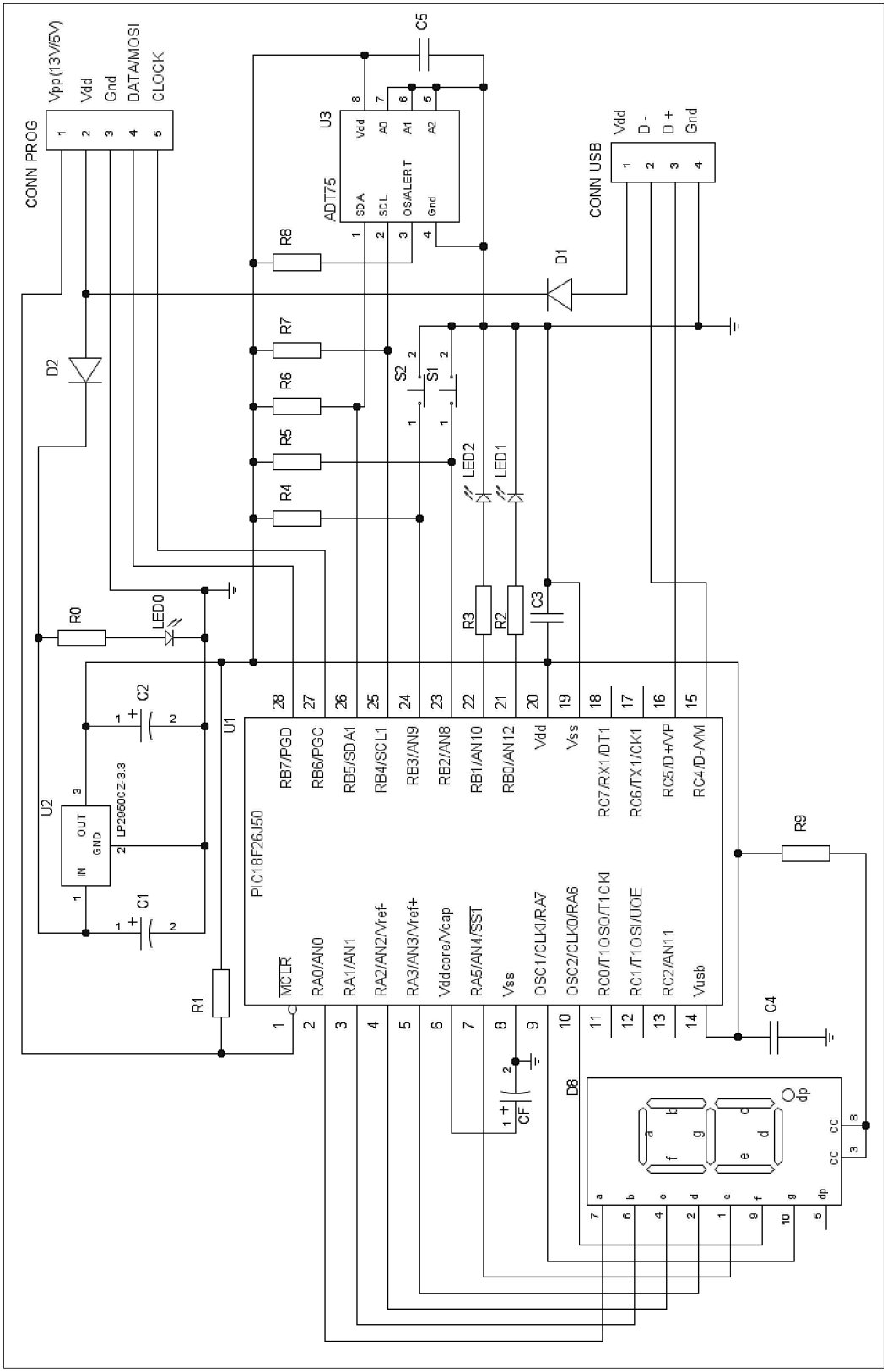
4.3.1 Vstup a výstup

Všechny součásti popsané výše plní nějakou funkci a vzhledem k mikrokontroléru to jsou buď vstupy nebo výstupy. Pod vstupem si můžeme představit např. různá čidla nebo senzory (pohybu, teploty, tlaku, magnetického pole atd.), fotocitlivé prvky (fototranzistory, IR přijímače atd.) apod. V našem případě jsou vstupy digitální teploměr U3 a 2 mikrospínače S1 a S2. Podobně můžeme přemýšlet o výstupech, které mohou být např. různé zobrazovací prvky (LED, LED displeje, LCD displeje atd.), akustické měniče, reléové spínací prvky apod. V našem případě jde o 2 svítivé diody LED1, LED2 a sedmisegmentový LED displej D8. Takovéto rozdělení nám pomůže identifikovat směr komunikace přes USB rozhraní. Vstup bude zřejmě znamenat komunikaci směrem od mikrokontroléru k počítači a výstup značí směr od počítače k mikrokontroléru.

4.3.2 Digitální teploměr

Rád bych se zastavil u digitálního teploměru, který je nejsložitější periférií mikrokontroléru v tomto ukázkovém příkladu. Použil jsem teplotní čidlo s číslicovým výstupem ADT75ARMZ, což je kompletní monitorovací systém teploty v 8vývodovém pouzdru. Obsahuje 12bitový analogově-digitální převodník, který umožňuje rozlišení až 0,0625 °C. Dosahuje přesnosti maximálně ± 3 °C při napájecím 3,0 – 3,6 V. Může být napájen napětím 3,0 – 5,5 V. Při napětí 3,3 V odebírá proud pouhých 200 μA. Je kompatibilní s teplotními čidly LM75 a AD7416 z hlediska počtu a rozložení vývodů i z hlediska vnitřních registrů. Dá se nahradit i teplotním senzorem DS1621, který má stejný počet i rozložení vývodů, ale liší se vnitřními registry. Všechny tyto integrované obvody mohou komunikovat se svým okolím prostřednictvím dvoulinkové sériové sběrnice I^2C, kterou podporuje i mikrokontrolér PIC18F26J50, což velice zjednodušuje práci.

Ze začátku jsem zkoušel zprovoznit teplotní senzor DS18B20, který má pouze 3 vývody a dosahuje vyšší přesnosti (± 0,5 °C). Pro přenos dat však používá jednovodičovou sběrnici 1-Wire, která vyžaduje vysokou přesnost časování při přenosu dat. Mikrokontrolér PIC18F26J50 tuto sběrnici nepodporuje, takže je nutné napsat vlastní program, který na některém vývodu emuluje sběrnici 1-Wire, což se značně komplikuje. Tento problém by se dal vyřešit pomocí převodníku sběrnic I^2C na 1-Wire DS2482.



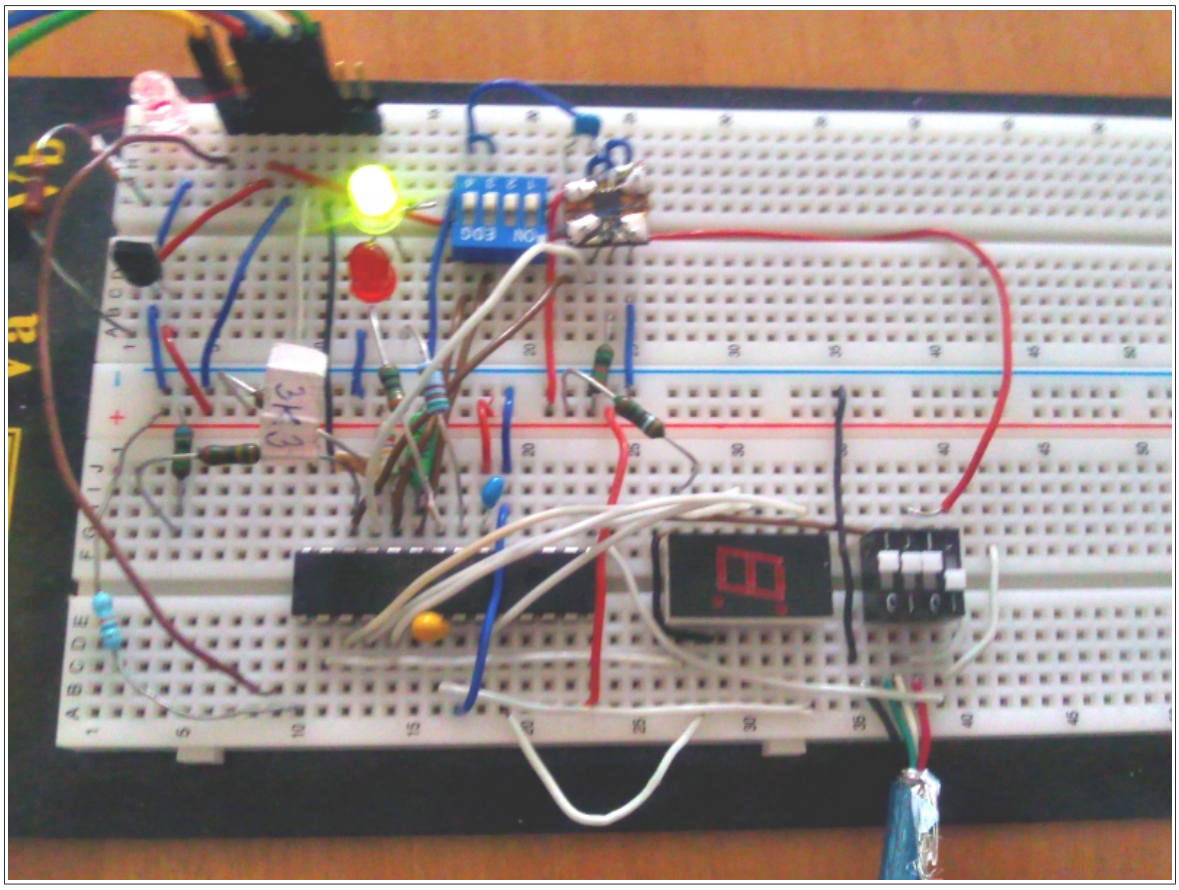
Obrázek 11: Zapojení mikrokontroléru PIC18F26J50

Zapojení teplotního senzoru ADT75

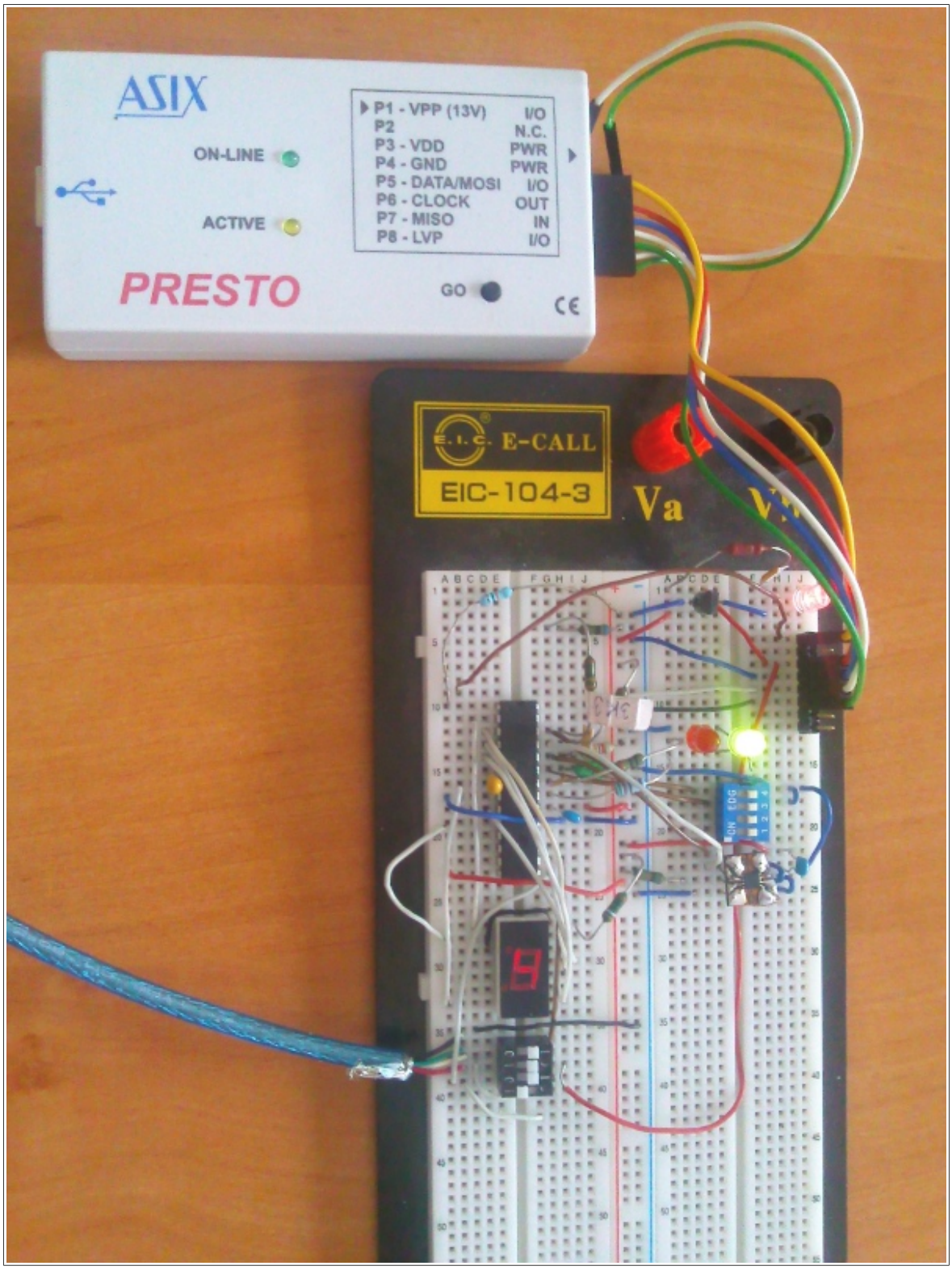
Vraťme se ale k teploměru ADT75. Na obrázku 11 je mimo jiné zachyceno jeho typické zapojení. Na vývodu V_{DD} je přivedeno napájecí napětí, v našem případě 3,3 V. Je k němu připojen i oddělovací kondenzátor C5, který by měl být keramický s kapacitou 100 nF pro napětí minimálně 20 V a umístěný do 6 mm od součástky. Vývody A0, A1 a A2 slouží k nastavení adresy zařízení. Celková délka adresy je 7 bitů, z čehož nejvyšší 4 bity jsou nastaveny přímo v integrovaném obvodu na hodnotu 1001 a spodní 3 bity jsou určeny logickou úrovní napětí přivedeného na vývody A0, A1 a A2. V našem případě jsou uzemněny (přivedeny na záporný pól zdroje), takže nastavují nejnižší 3 bity na hodnotu 000. Výsledná adresa je tedy binárně zapsána následovně: 1001000. Vývody SDA a SCL slouží pro komunikaci s mikrokontrolérem přes sběrnici I^2C. Paralelně k těmto vývodům je přivedeno napětí 3,3 V přes rezistory R6 a R7 (angl. pull-up resistors), které mají odpor 10 kΩ a udržují tak sběrnici v klidu na logická jedničce. Vývod OS/ALERT plní účel termostatu. Pokud se teplota zvýší nad resp. klesne pod nastavený limit, aktivuje se výstup OS/ALERT. Tato funkcionalita však není předmětem této diplomové práce, takže v případě zájmu pouze odkazuji čtenáře na dokumentaci ADT75 – literaturu [11]. Vývod Gnd je přiveden na záporný pól zdroje (uzemněn).

4.4 Zapojení na nepájivém poli

Podle schématu na obrázku 11 můžeme propojit všechny součástky na nepájivém poli, jak ukazuje fotografie 12. Toto zapojení však zcela jistě nebude fungovat, protože mikrokontrolér PIC18F26J50 není ještě naprogramován, tedy neobsahuje žádný program, kterým by mohl obsluhovat své vstupně-výstupní obvody. Nezbývá nám nic jiného než připojit k obvodu programátor (viz obrázek 13) a pustit se do samotného psaní obslužného kódu mikrokontroléru tzv. firmwaru (viz další kapitola 5).



Obrázek 12: Zapojení mikrokontroléru na nepájivém poli



Obrázek 13: Zapojení mikrokontroléru na nepájivém poli s připojeným programátorem

5 Firmware mikrokontroléru

V kapitole 2.2.1 jsme si popsali, jaké vývojové nástroje máme k dispozici. Současná doba nám nabízí velice silné nástroje, které nám umožňují psát program mikrokontroléru v jazyce C, který je mnohem přehlednější a čitelnější než program psaný v Assembleru, ve kterém se musely všechny programy psát v minulosti. Je pravda, že pomocí Assembleru můžeme získat efektivnější a kratší výsledný kód, ale v dnešní době, kdy už se nehledí na každý drahocenný byte a rychlost mikrokontrolérů převyšuje rychlost mikroprocesorů prvních stolních počítačů, je tento nedostatek irelevantní. Pojďme si tedy ukázat, jak napsat nějaký jednoduchý program a poté popíši oživení demonstrační aplikace.

5.1 Jak začít programovat mikrokontrolér

Pro psaní programů mikrokontroléru budeme potřebovat počítač s USB rozhraním a s operačním systémem, ideálně volně šiřitelnou distribucí Linuxu, v mém případě Ubuntu Oneiric (11.10). Dále pak vývojové prostředí, tedy MPLAB X IDE a kompilátor MPLAB C18 v3.40.

5.1.1 První program

Před samotným programováním bych doporučil postupovat podle tutoriálu firmy Microchip (literatura [12]), který je sice napsán v angličtině, ale velice dobře popisuje, jak přistupovat k portům mikrokontroléru, jak ovládat jednotlivé vývody a jak pracovat s datovými typy. Vhodné je také podívat se na dokumentaci a příklady dodávané s kompilátorem MPLAB C18, které lze najít v následujících adresářích resp. souborech (pokud je kompilátor nainstalován ve výchozím adresáři `/opt/microchip/mplabc18/v3.40`):

- `/opt/microchip/mplabc18/v3.40/MPLAB-C18-README.html`
- `/opt/microchip/mplabc18/v3.40/doc`
- `/opt/microchip/mplabc18/v3.40/example`

Uvedu zde jeden příklad (viz zdrojový kód 4), který jsem mírně poupravil podle ukázkového zdrojového kódu `main.c` v adresáři:

```
/opt/microchip/mplabc18/v3.40/examples/getting_started/program3
```

Je to jednoduchý program, který pouze rozblikává LED. První řádek (`#include <p18cxxx.h>`) přilinkovává knihovnu `p18cxxx.h`, která podle daného parametru (při kompilaci) vybere knihovnu odpovídající požadovanému mikrokontroléru. Pokud např. použijeme následující příkaz, který zkompiluje ukázkový program:

```
/opt/microchip/mplabc18/v3.40/bin/mcc18 -p 18f26j50 \
-I /opt/microchip/mplabc18/v3.40/h main.c
```

bude vybrána knihovna `p18f26j50.h`, která odpovídá mikrokontroléru PIC18F26J50. Další řádek

(`#pragma config WDTEN = OFF`), deaktivuje tzv. watchdog (systém, který po určitém čase restartuje program, pokud mikrokontrolér neprovede požadovanou operaci po určitou předem nastavenou dobu). Následuje funkce `void delay(void)` generující čekání. Hlavní funkce `void main(void)` nastavuje v prvním řádku všechny bity (vývody) portu B na výstup (viz další kapitola 5.1.2). Dále obsahuje hlavní nekonečnou smyčku, která vždy zhasne (resetuje) všechny LED připojené k portu B, počká nějakou dobu, potom rozsvítí LED na portu B (vývody RB1, RB3, RB4 a RB6) a opět počká krátkou chvíli.

```
#include <p18cxxx.h>

#pragma config WDTEN = OFF

void delay(void)
{
    int i;
    for (i = 0; i < 10000; i++) ;
}

void main(void)
{
    // Make all bits on the Port B (LEDs) output bits.
    // If bit is cleared, then the bit is an output bit.
    TRISB = 0;

    while (1)
    {
        LATB = 0;       // Reset the LEDs
        delay();        // Delay so human eye can see change
        LATB = 0x5A;    // Light the LEDs
        delay();        // Delay so human eye can see change
    }
}
```

Zdrojový kód 4: Ukázkový program – blikání LED – převzato z [12]

5.1.2 Vstupně-výstupní porty a registry TRIS, PORT, LAT

Rád bych se zmínil o vlastnostech vstupně-výstupních portů. Každý port je namapován na několik registrů, které daný port konfigurují a ovládají. Port může být nastaven jako vstupní, nebo výstupní, od čehož se odvíjí, zda budou z portu data čtena, nebo jestli budou data na port zapisována. Následující 3 registry slouží k nastavení daného portu a k operacím nad portem:

- TRIS: nastavuje směr přenosu dat na portu (0 = výstup, 1 = vstup)
- PORT: čte úrovně na vývodech portu

- LAT: udržuje číslicové hodnoty, které mají být přítomny na portu v případě, že je port nastaven jako výstupní

Jinými slovy, registr PORT se používá pro vstupní operace, registr LAT pro výstupní operace a registr TRIS nastavuje port na vstup, nebo výstup.

Každý port lze ovládat i po jednom bitu pomocí předdefinovaných struktur:

- TRISXbits
- PORTXbits
- LATXbits

kde X je písmeno portu (A, B, nebo C). Např. příkaz:

- `TRISAbits.TRISA5 = 1;` nastaví pin RA5 portu A na vstup
- `char x = PORTBbits.RB2;` nastaví hodnotu x na hodnotu pinu RB2 portu B (0 nebo 1)
- `LATCbits.LATC7 = 0;` nastaví logickou úroveň 0 na výstupu RC7 portu C

5.1.3 Přenesení firmwaru do mikrokontroléru

Zkompilovaný program je jen jednou součástí výsledné aplikace, kterou potřebujeme přenést na mikrokontrolér. Další součástí je knihovna mikrokontroléru. Abychom dostali výsledný binární kód, musíme použít sestavovací program (angl. linker), který spojí všechny objektové soubory vygenerované překladačem s potřebnými knihovnami dohromady (tzv. linkování). K tomu použijeme následující příkaz vztahující se k ukázkovému příkladu zdrojového kódu 4:

```
/opt/microchip/mplabc18/v3.40/bin/mplink -p18f26j50 -w \
-u_CRUNTIME -l /opt/microchip/mplabc18/v3.40/lib -o main.cof main.o
```

Teprve výsledný binární soubor s příponou .hex můžeme přenést na mikrokontrolér (říkáme, že naprogramujeme mikrokontrolér) pomocí programátoru. Na obrázcích 10 a 11 je znázorněno, jak připojit programátor do obvodu.

5.1.4 Další informace

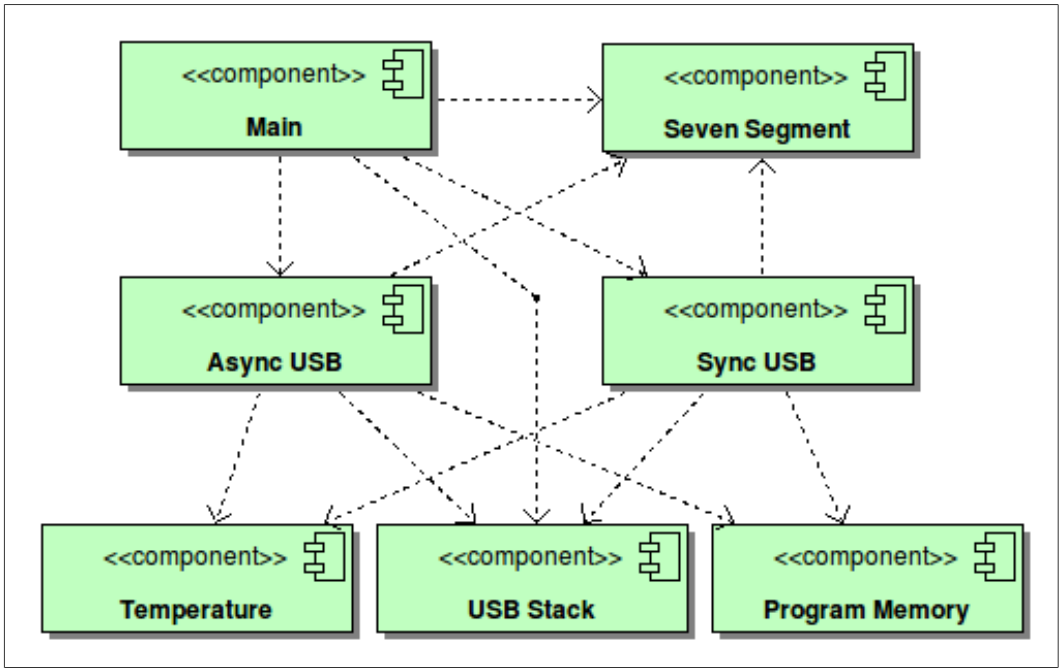
Výše uvedený ukázkový příklad je jen krátkým návodem, jak začít programovat. Pro více informací doporučuji, přečíst si dokumentaci a popis knihoven kompilátoru MPLAB C18, které lze najít na stránkách firmy Microchip (literatura [13]). Informace o tom, jak zprovoznit a obsluhovat programátor, v mém případě ASIX PRESTO, lze najít v literatuře [14].

5.2 Firmware demonstrační aplikace

Program demonstrační aplikace, který jsem vyvinul, je už více komplexní, takže jej bylo nutné rozdělit na několik menších modulů, které řeší specifické funkce. Jejich seznam je následující:

- Hlavní modul (zkráceně Main)
- USB modul (angl. USB stack, nebo také MCHPFSUSB framework – zkráceně USB Stack)
- Modul synchronní komunikace přes USB (zkráceně Sync USB)
- Modul asynchronní komunikace přes USB (zkráceně Async USB)
- Modul obsluhy sedmisegmentového displeje (zkráceně Seven Segment)
- Modul čtení teploty (zkráceně Temperature)
- Modul pro manipulaci s programovou pamětí (zkráceně Program Memory)

Na obrázku 14 jsou znázorněny závislosti mezi jednotlivými moduly.



Obrázek 14: Diagram komponent – moduly firmwaru

5.2.1 Konfigurační soubory

Nedílnou součástí celé aplikace jsou konfigurační soubory, které obsahují informace o použitých periféríích a definice vyžadované USB modulem. Patří mezi ně:

- `HardwareProfile.h`: obsahuje informace o všech dostupných hardwarových profilech a slouží k výběru jednoho z nich
- `HardwareProfile – PIC18F26J50.h`: obsahuje definice vyžadované USB modulem, informace o používaném mikrokontroléru PIC18F26J50 a definice specifické pro aplikaci (společné aliasy pro vývody a porty, identifikace periférií atd.)
- `usb_config.h`: obsahuje nastavení USB modulu (velikost vyrovnávací paměti pro koncové body, počet koncových bodů a jejich aliasy atd.), názvy příkazů používaných při USB komunikaci a jejich kódy a adresy paměťového prostoru programu obsahující jedinečnou identifikaci koncového zařízení
- `usb_descriptors.c`: obsahuje USB deskriptory včetně sériového čísla, které slouží k jedinečné identifikaci koncového zařízení

5.2.2 Hlavní modul

Všechny moduly jsou buď přímo nebo nepřímo využívány hlavním modulem – souborem `main.c`. Jeho součásti jsou následující:

- konfigurace vnitřních registrů mikrokontroléru
- přemapování vektorů přerušení a obslužné funkce přerušení
- hlavní funkce `void main(void)` obsahující hlavní nekonečnou smyčku programu
- funkce pro inicializaci systému
- funkce obsluhující vstupně-výstupní operace
- prázdné definice funkcí zpětného volání pro obsluhu USB událostí (angl. USB Callback Functions)

Po uvedení systému do provozu (přivedením napájecího napětí k mikrokontroléru) se spustí program – je zavolána hlavní funkce `main()` (viz zdrojový kód 5), která pouze zavolá funkci pro inicializaci systému `InitializeSystem()`, poté aktivuje USB modul zavoláním funkce `USBDeviceAttach()` (oznámí USB kořenovému uzlu, že je ke sběrnici připojeno koncové zařízení) a vstoupí do hlavní nekonečné smyčky. V této smyčce se volá pouze jedna funkce `ProcessIO()`, která obsluhuje vstupně-výstupní operace tím, že volá funkce ostatních modulů (viz zdrojový kód 6). Zakomentováním určitých řádků (např. řádek: `handleUSBAsync();`) můžeme deaktivovat některé moduly (např. modul asynchronní komunikace přes USB). Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `main.c`.

```
void main(void)
{
    InitializeSystem();

    #if defined(USB_INTERRUPT)
        USBDeviceAttach();
    #endif

    while(1)
    {
        #if defined(USB_POLLING)
            // Check bus status and service USB interrupts.
            // Interrupt or polling method.
            // If using polling, must call this function periodically.
            USBDeviceTasks();
        #endif

      // Application-specific tasks.
      // Application related code may be added here,
      // or in the ProcessIO() function.
      ProcessIO();
    }
}
```

Zdrojový kód 5: Hlavní smyčka programu pro mikrokontrolér

```
void ProcessIO(void)
{
    unsigned char syncResult;

    // Display the date on the seven segment
    displayDate();

    // Blink the LEDs according to the USB device status,
    // but only do so if the PC application isn't connected and
    // controlling the LEDs.
    if (blinkStatusValid)
        BlinkUSBStatus();

    // User Application USB tasks below.
    // Note: The user application should not begin attempting
    // to read/write over the USB until after the device has been fully
    // enumerated. After the device is fully enumerated,
    // the USBDeviceState will be set to "CONFIGURED_STATE".
    if ((USBDeviceState < CONFIGURED_STATE)
        || (USBSuspendControl == 1)) return;

    // handle the synchronous communication
    syncResult = handleUSBSync();

    // display the synchronous communication error on the seven segment
    if (syncResult != SYNC_RETURN_OK)
        date[12] = convertToChar(syncResult);

    // handle the asynchronous communication
    handleUSBAsync();
}
```

Zdrojový kód 6: Obsluha vstupně-výstupních operací v programu mikrokontroléru

5.2.3 USB modul (angl. USB stack, nebo také MCHPFSUSB framework)

Firma Microchip dodává tzv. aplikační knihovny, které lze stáhnout z internetových stránek firmy [15] v jediném souboru, podle toho, jaký operační systém používáte. Já jsem použil verzi pro Linux:

```
microchip-application-libraries-v2011-07-14-beta-linux-
installer.run
```

Mezi knihovny patří mimo jiné i celý USB framework, který lze po menších úpravách použít pro daný mikrokontrolér (viz příloha A). Následující soubory musíme upravit pro potřeby demonstrační aplikace: `HardwareProfile.h`, `HardwareProfile - PIC18F46J50 PIM.h`, `main.c`, `usb_config.h`, `usb_descriptors.c` (viz předchozí kapitoly 5.2.1 a 5.2.2).

5.2.4 Modul synchronní komunikace přes USB

Do tohoto modulu patří pouze dva soubory:

- `include/usb_communication_sync.h`
- `usb_communication_sync.c`

Hlavičkový soubor `usb_communication_sync.h` obsahuje deklarace funkcí a soubor `usb_communication_sync.c` definice funkcí:

- `void initHandlerSync(void)`: inicializuje synchronní modul
- `unsigned char handleUSBSync(void)`: obsluhuje synchronní komunikaci přes USB
  - Směr z PC na koncové zařízení: nastavení a získání ID koncového zařízení, čtení několika bytů dat z mikrokontroléru, obsluha LED, získání teploty
  - Směr z koncového zařízení na PC: získání aktuálního data (den, měsíc, rok), který se pak zobrazuje na sedmisegmentovém displeji
- `void setId(BYTE id1, BYTE id2, BYTE id3, BYTE id4, BYTE id5, BYTE id6)`: nastavuje část sériového čísla pro jedinečnou identifikaci koncového zařízení

Z ukázek zdrojových kódů 7, 8 a 9 můžeme vyčíst, jak funguje obsluha synchronní komunikace. Zdrojový kód 7 popisuje odeslání požadavku (získání aktuálního data) na kořenový uzel (PC aplikace) a zdrojový kód 8 příjem odpovědi. Jde tedy o synchronní komunikaci směrem z koncového zařízení k PC aplikaci. Teoreticky by měl program mikrokontroléru po odeslání požadavku čekat, až dorazí odpověď, např. ve smyčce kontrolovat, jestli už odpověď dorazila. To by však bylo zbytečné plýtvání prostředků, proto jsem navrhl program tak, že se po odeslání požadavku pouze nastaví příznak `waitingForResponse` na logickou jedničku (viz zdrojový kód 7) a vykonávání programu pokračuje. Dále se kontroluje, zda je tento příznak nastaven a jestli dorazila odpověď v definovaném časovém limitu (viz zdrojový kód 8). Pokud ano, nastaví se nový datum, smaže se příznak `waitingForResponse` a počítadlo časového limitu.

Na zdrojovém kódu 9 pak můžeme vidět, jakým způsobem koncové zařízení odpovídá na požadavky PC aplikace (synchronní komunikaci směrem z PC aplikace ke koncovému zařízení). Pokud dorazí nějaký požadavek/příkaz na koncové zařízení, je podle prvního bytu ve vyrovnávací paměti koncového bodu rozhodnuto, o jaký typ požadavku jde a jakým způsobem bude obsloužen. V některých případech může dojít pouze ke změně vnitřních registrů mikrokontroléru (ve zdrojovém kódu 9: nastavení ID – příkaz `COMMAND_SET_ID`), nebo může být navíc odeslána odpověď zpět na kořenový uzel (ve zdrojovém kódu 9: získání ID – příkaz `COMMAND_GET_ID`).

Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `usb_communication_sync.c`.

```
int returnValue = SYNC_RETURN_OK;

// if the pushbutton is pressed send the Get Date command
// to the host PC
if (sw3 == 0)
{
    syncINPacket[0] = COMMAND_GET_DATE;

    // Now check to make sure no previous attempts to send data
    // to the host are still pending.
    if (!USBHandleBusy(syncUSBGenericInHandle))
    {
        // The endpoint was not "busy", therefore it is safe
        // to write to the buffer and arm the endpoint.
        syncUSBGenericInHandle = USBGenWrite(USBGEN_EP_SYNC,
            (BYTE*)&syncINPacket, USBGEN_EP_SIZE);

        // set the waiting flag
        waitingForResponse = SYNC_WAITING;
    }
}
```

Zdrojový kód 7: Synchronní komunikace v programu mikrokontroléru – odeslání požadavku na kořenový uzel

```
if (waitingForResponse == SYNC_WAITING)
{
    // check if the response arrived or the timeout has been reached
    if (counter++ > timeout)
    {
        // clear the waiting flag
        waitingForResponse = SYNC_NOT_WAITING;
        counter = 0;

        // Error - cannot get the date from the host
        date[11] = 'E';

       return SYNC_RETURN_TIMEOUT;
    }

    if (!USBHandleBusy(syncUSBGenericOutHandle)
        && syncOUTPacket[0] == COMMAND_GET_DATE)
    {
        // set the date
        ...
        // no error - the date has been acquired from the host
        date[11] = '-';

        // Re-arm the OUT endpoint for the next packet
        syncUSBGenericOutHandle = USBGenRead(USBGEN_EP_SYNC,
            (BYTE*)&syncOUTPacket, USBGEN_EP_SIZE);

        // clear the waiting flag
        waitingForResponse = SYNC_NOT_WAITING;
        counter = 0;
        date[12] = '-';
    }
    else
    {
        // Error - cannot get the date from the host
        date[11] = 'E';
    }
}
```

Zdrojový kód 8: Synchronní komunikace v programu mikrokontroléru – příjem odpovědi

```
// Check if the endpoint has received any data from the host.
if (!USBHandleBusy(syncUSBGenericOutHandle))
{
    // Data arrived, check what kind of command might be
    // in the packet of data.
    switch(syncOUTPacket[0])
    {
        case COMMAND_SET_ID:
        {
            // check if another event arived while waiting
            // for the response
            if (waitingForResponse == SYNC_WAITING)
                returnValue = SYNC_RETURN_ANOTHER_EVENT;
            ...
            setId(syncOUTPacket[1], syncOUTPacket[2],
                    syncOUTPacket[3], syncOUTPacket[4],
                    syncOUTPacket[5], syncOUTPacket[6]);
            break;
        }
        case COMMAND_GET_ID:
        {
            ...
            // Echo back to the host PC the command we are fulfilling
            // in the first byte. In this case, the Get ID command.
            syncINPacket[0] = COMMAND_GET_ID;

            // populate the USB device serial number id
            readProgramMemoryBytes(USER_DATA_ROM_ADDRESS_START +
                SERIAL_NUMBER_OFFSET * 2, syncINPacket, 1, 6);

            if (!USBHandleBusy(syncUSBGenericInHandle))
            {
                syncUSBGenericInHandle = USBGenWrite(USBGEN_EP_SYNC,
                    (BYTE*)&syncINPacket, USBGEN_EP_SIZE);
            }
            break;
        }
        ...
    }
    // Re-arm the OUT endpoint for the next packet:
    syncUSBGenericOutHandle = USBGenRead(USBGEN_EP_SYNC,
        (BYTE*)&syncOUTPacket, USBGEN_EP_SIZE);
}
return returnValue;
```

Zdrojový kód 9: Synchronní komunikace v programu mikrokontroléru

5.2.5 Modul asynchronní komunikace přes USB

Do tohoto modulu patří pouze dva soubory:

- `include/usb_communication_async.h`
- `usb_communication_async.c`

Hlavičkový soubor `usb_communication_async.h` obsahuje deklarace funkcí a soubor `usb_communication_async.c` definice funkcí:

- `void initHandlerSync(void)`: inicializuje asynchronní modul
- `unsigned char handleUSBSync(void)`: obsluhuje asynchronní komunikaci přes USB
  - Směr z PC na koncové zařízení: získání teploty
  - Směr z koncového zařízení na PC: odeslání události, která byla vyvolána na koncovém zařízení (stisknutí tlačítka, alarm)

Asynchronní komunikace je v programu mikrokontroléru implementovaná úplně stejně, jako synchronní komunikace. Použil jsem pro ni však jiné čísla koncových bodů (angl. endpoints), aby nedocházelo k nechtěnému přepisování vyrovnávacích pamětí. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `usb_communication_async.c`.

5.2.6 Modul obsluhy sedmisegmentového displeje

Do tohoto modulu patří pouze dva soubory:

- `include/seven_segment_output.h`
- `seven_segment_output.c`

Hlavičkový soubor `seven_segment_output.h` obsahuje deklarace funkcí a soubor `seven_segment_output.c` definice funkcí:

- `char getSymbol(unsigned char codedSymbol)`: vrací číslo nebo znak čitelný člověkem, který se zobrazuje na sedmisegmentovém displeji
- `unsigned char getCodedSymbol(char symbol)`: vrací binární kód čísla nebo znaku čitelného člověkem, který se posílá na port mikrokontroléru. Tabulka 9 v příloze B obsahuje převody mezi znaky (včetně čísel) a odpovídajícími hodnotami kódu pro port.
- `void displayLetter(void)`: zobrazuje jeden znak na sedmisegmentovém displeji, slouží pro testovací účely
- `void displayDate(void)`: zobrazuje jeden znak data na sedmisegmentovém displeji. Celý datum má formát: DD-MM-YYYY---, kde D je cifra dne, M je cifra měsíce a Y je cifra roku.
- `unsigned char convertToChar(int value)`: převádí celé číslo v rozmezí 0 – 15 na znak zobrazitelný na sedmisegmentovém displeji (0 – 9, A, b, C, d, E, F). Je to v podstatě převod desítkového čísla na šestnáctkové s ohledem na možnosti zobrazení sedmisegmentového displeje.

5.2.7 Modul čtení teploty

Do tohoto modulu patří soubory:

- `include/temperature_input.h`
- `include/adt75.h`
- `include/ds1621.h`
- `temperature_input.c`
- `adt75.c`
- `ds1621.c`

Hlavičkový soubor `temperature_input.h` obsahuje deklaraci funkce a soubor `temperature_input.c` definici funkce:

```
void getTemperature(unsigned char *temperature),
```

která získává teplotu z nakonfigurovaného teplotního senzoru (ADT75, nebo DS1621). Získaná teplota se uloží do parametru `temperature`, což je pole o dvou prvcích, kde první prvek obsahuje vyšší byte a druhý prvek obsahuje nižší byte hodnoty teploty. Hlavičkové soubory `adt75.h` a `ds1621.h` obsahují deklarace funkcí a soubory `adt75.c` a `ds1621.c` definice funkcí:

```
int I2CGetTempADT75(void)
```
, resp.
```
int I2CGetTempDS1621(void),
```

které jsou specifické pro daný teplotní senzor. Každá funkce obsahuje rutinu pro získání teploty z čidla pomocí přenosu dat přes I^2C sběrnici.

Registry teplotního senzoru ADT75

ADT75 obsahuje 6 registrů: 4 datové registry, registr ukazatele na adresu (angl. address pointer register) a registr pro jednorázové čtení nebo zápis (angl. one-shot register). Konfigurační registr (angl. configuration register) je jako jediný 8bitový a ostatní registry jsou 16bitové. Registr hodnoty teploty (angl. temperature value register) je jediný datový registr, který je pouze pro čtení. Do tohoto registru se ukládá hodnota teploty naměřené vnitřním teplotním senzorem. Tato 16bitová hodnota má formát dvojkového doplňku, kde nejvyšší bit určuje znaménko. Struktura registru je znázorněna v tabulce 4 a reprezentace některých hodnot jsou uvedeny v tabulce 5, ze které lze také vyčíst, že jednomu bitu odpovídá hodnota teploty 0,062 5 °C.

MSB **LSB**

D15	**D14**	**D13**	**D12**	**D11**	**D10**	**D9**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
0	0	0	0	0	0	0	0	0	0	0	0	N/A	N/A	N/A	N/A

Tabulka 4: Registr hodnoty teploty teplotního senzoru ADT75 – převzato z [11]

Teplota	Reprezentace teploty (bity D15 – D4)	Reprezentace teploty (hexadecimálně)
–55 °C	1100 1001 0000	0xC90
–50 °C	1100 1110 0000	0xCE0
–25 °C	1110 0111 0000	0xE70
–0,062 5 °C	1111 1111 1111	0xFFF
0 °C	0000 0000 0000	0x000
+0,062 5 °C	0000 0000 0001	0x001
+10 °C	0000 1010 0000	0x0A0
+25 °C	0001 1001 0000	0x190
+50 °C	0011 0010 0000	0x320
+75 °C	0100 1011 0000	0x4B0
+100 °C	0110 0100 0000	0x640
+125 °C	0111 1101 0000	0x7D0

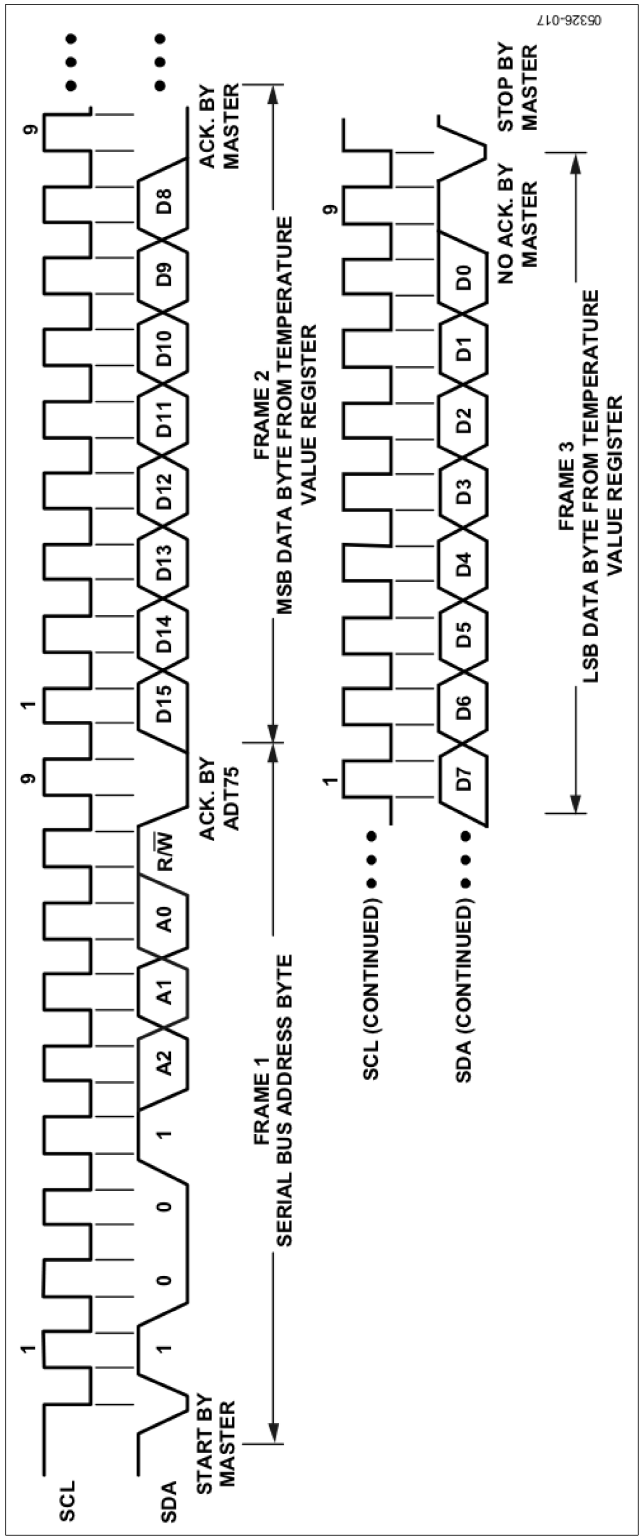
Tabulka 5: 12bitový formát teploty teplotního senzoru ADT75 – převzato z [11]

Komunikace s teplotním senzorem ADT75

Jak už jsem popsal v kapitole 4.3.2, teplotní senzor ADT75 komunikuje se svým okolím přes sériové rozhraní SMBus/I²C. Zařízení připojená ke sběrnici se dělí na řídící (angl. master) a řízená (angl. slave). V našem případě je vždy řídícím zařízením mikrokontrolér a řízeným zařízením je vždy teplotní senzor. Komunikaci zahajuje pouze mikrokontrolér tím, že pošle na sběrnici adresu cílového zařízení, které se tímto aktivuje. Poté pošle jeden bit, který rozhoduje o směru přenosu dat:

- 0: data se budou posílat na teplotní senzor (angl. write)
- 1: data se budou číst z teplotního senzoru (angl. read)

Následuje jeden bit potvrzení od teplotního senzoru a po něm probíhá samotný přenos dat. Pokud chceme zapisovat/číst data do/z ADT75, musí být nejprve vybrán jeden z registrů. Při startu teplotního čidla je registr ukazatele na adresu nastaven na hodnotu 0, což odpovídá ukazateli na registr hodnoty teploty, takže není nutné přenastavovat tento ukazatel na registr v případě, že chceme pouze číst teplotu. Na obrázku 15 vidíme ukázku časového diagramu, který znázorňuje čtení z registru hodnoty teploty, kdy se nejdříve posílá vyšší byte (bity D15 – D8) a za ním nižší byte (bity D7 – D0). Ukázka zdrojového kódu 10 popisuje, jak inicializovat I²C sběrnici. K tomu je nutné přilinkovat knihovnu `i2c.h`. Zdrojový kód 11 pak ukazuje sled příkazů a volání funkcí, díky kterým lze komunikovat s teplotním čidlem přes I²C sběrnici. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `adt75.c` resp. `ds1621.c`. Podobným způsobem lze ovládat i teplotní senzor DS1621. Doporučuji také, přečíst si katalogové listy součástek – literatury [11 a 16].



Obrázek 15: Čtení dat z registru hodnoty teploty z teplotního senzoru ADT75 – převzato z [11]

```
#include <i2c.h>

...

unsigned char sync_mode, slew, dataH, dataL, status;

CloseI2C();     // close i2c if was operating earlier

//---- INITIALISE THE I2C MODULE FOR MASTER MODE WITH 100KHz ----------
sync_mode = MASTER;
slew = SLEW_OFF;

OpenI2C(sync_mode, slew);

// FOSC = 8 MHz, SYNC = 0, BRGH = 1, BRG16 = 1 or SYNC = 1, BRG16 = 1
// Actual Baud Rate (K) = 117.647, SPBRG value (decimal) = 16
SSPADD = 16;

// check for bus idle condition in multi master communication
IdleI2C();
```

Zdrojový kód 10: Komunikace s ADT75 přes sběrnici I^2C – inicializace sběrnice

```
//---- RESTART I2C COMMUNICATION -------------------------------------
RestartI2C();
IdleI2C();

// write the address of the device for communication
// read any previous stored content in buffer
// to clear buffer full status
dataH = SSPBUF;
do
{
    status = WriteI2C(ADT75_ADDRESS | 0x01); // write the slave address
    if (status == -1)       // check if bus collision happened
    {
        // upon bus collision detection clear the buffer,
        dataH = SSPBUF;
        SSPCON1bits.WCOL = 0;   // clear the bus collision status bit
    }
}
while (status != 0);        // write untill successful communication
// R/W BIT IS '1' FOR READ FROM SLAVE


// Recieve data from slave
dataH = ReadI2C();          // recieve data from slave

NotAckI2C();    // send the end of transmission signal through nack
while (SSPCON2bits.ACKEN != 0); // wait till ack sequence is complete

// Recieve data from slave
dataL = ReadI2C();              // recieve data from slave

NotAckI2C();    // send the end of transmission signal through nack
while (SSPCON2bits.ACKEN != 0); // wait till ack sequence is complete


//---- CLOSE I2C COMMUNICATION ---------------------------------------
CloseI2C();                     // close I2C module
```

Zdrojový kód 11: Komunikace s ADT75 přes sběrnici I²C – přenos dat

5.2.8 Modul pro manipulaci s programovou pamětí

Poslední modul obsahuje pouze dva soubory:

- `include/program_memory_manipulation.h`
- `program_memory_manipulation.c`

Hlavičkový soubor `program_memory_manipulation.h` obsahuje deklarace funkcí a soubor `program_memory_manipulation.c` definice funkcí:

- `void readProgramMemoryBytes(unsigned short long address, BYTE* data, unsigned int dataOffset, unsigned int n)`:
  čte `n` bytů z programové paměti začínajících na adrese `address` a ukládá je do pole `data` od pozice `dataOffset`
- `void eraseProgramMemory(unsigned short long address)`: smaže jeden blok programové paměti o velikosti 1 024 bytů (512 slov) od adresy `address`, což je minimální počet bytů, které lze smazat
- `void writeToProgramMemory(unsigned short long address, WORD data)`:
  zapíše `data` o velikosti jednoho slova (2 byty) do programové paměti na adresu `address`
- `void rewriteProgramMemoryWord(unsigned short long address, WORD offset, WORD data)`: nejdříve smaže jeden blok programové paměti o velikosti 1 024 bytů (512 slov) od adresy `address` a pak zapíše `data` o velikosti jednoho slova (2 byty) do programové paměti na adresu `address + offset`
- `void rewriteProgramMemoryBlock(unsigned short long address, WORD* data, unsigned int n)`: nejdříve smaže jeden blok programové paměti o velikosti 1 024 bytů (512 slov) od adresy `address` a pak zapíše `data` o velikosti `n` slov (2n bytů) do programové paměti od adresy `address`

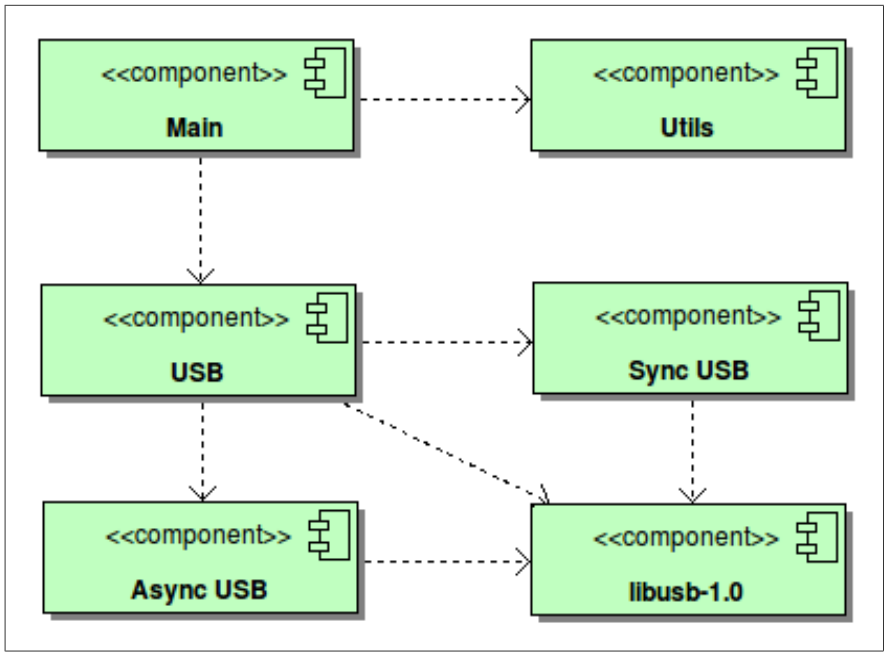
Pomocí těchto funkcí můžeme zapisovat do programové paměti data, která jsou dostupná i po odpojení napájecího napětí od mikrokontroléru. Díky nim lze uchovat např. sériové číslo, pomocí něhož můžeme jednoznačně rozlišovat koncová zařízení. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `program_memory_manipulation.c` popř. z dokumentace mikrokontroléru [10].

6 PC aplikace

Demonstrační PC aplikaci jsem napsal v jazyce C++, který přináší objektově orientovaný přístup. Podobně, jako firmware demonstrační aplikace popsaný v předchozí kapitole 5.2, je i PC aplikace rozdělena na moduly řešící specifické funkce:

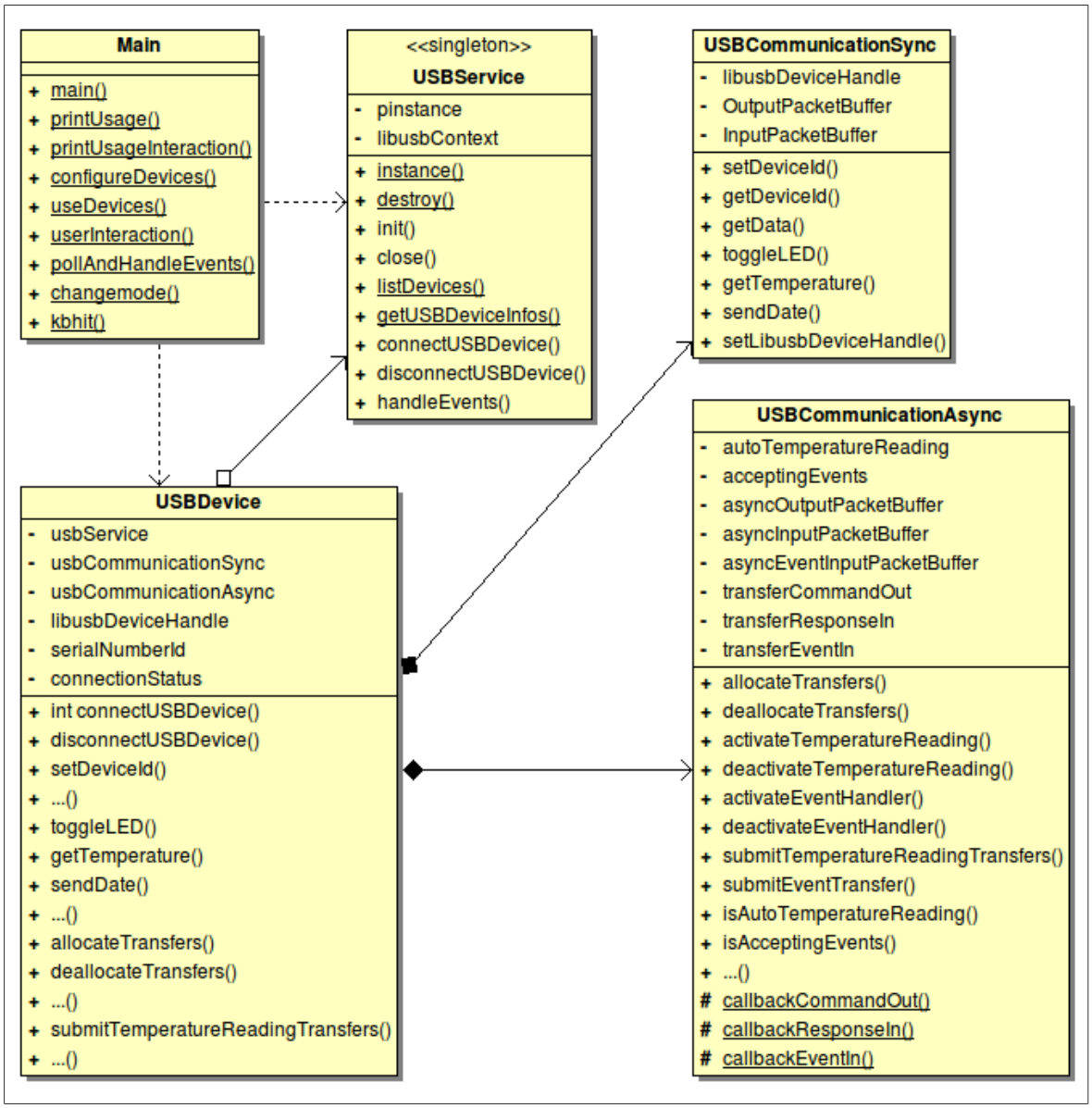
- Hlavní modul (zkráceně Main)
- USB modul (zkráceně USB)
- Modul synchronní komunikace přes USB (zkráceně Sync USB)
- Modul asynchronní komunikace přes USB (zkráceně Async USB)
- Modul podpůrných nástrojů (zkráceně Utils)

Na obrázku 16 jsou znázorněny závislosti mezi jednotlivými moduly včetně závislosti na knihovně libusb-1.0.



Obrázek 16: Diagram komponent – moduly PC aplikace

Třídní diagram na obrázku 17 pak ukazuje, jakým způsobem jsou provázané nejdůležitější třídy. Obsahuje mimo jiné element `Main`, který v samotném programu není implementován jako třída, ale jako soubor `main.cpp` obsahující funkce. Pro zpřehlednění je tento soubor zobrazen jako třída `Main` se statickými veřejnými metodami reprezentující funkce souboru `main.cpp`. Ostatní elementy jsou v programu implementovány jako třídy. Pro lepší čitelnost diagramu jsou některé nedůležité metody tříd nahrazeny symbolem `...()`.



Obrázek 17: Třídní diagram PC aplikace

6.1 Konstanty a definice

Celá aplikace využívá soubor `constants.h` (viz příloha F), který obsahuje konstanty, definice a jiné informace o používaném koncovém zařízení komunikujícím s aplikací přes USB. Např. velikost vyrovnávací paměti pro koncové body, aliasy koncových bodů, názvy příkazů používaných při USB komunikaci a jejich kódy, maximální počet koncových zařízení, s kterými může aplikace komunikovat.

6.2 Hlavní modul

Všechny moduly jsou buď přímo nebo nepřímo využívány hlavním modulem – souborem `main.cpp`. Jeho součástí je následující:

- hlavní funkce `int main(int argc, char* argv[])`, která podle argumentů zadaných uživatelem rozvětvuje chod programu na několik scénářů:
  - výpis nápovědy a návodu na použití aplikace
  - výpis seznamu všech dostupných koncových zařízení připojených k počítači přes USB
  - výpis seznamu všech dostupných relevantních koncových zařízení připojených k počítači přes USB, s kterými může aplikace komunikovat
  - konfigurační mód
  - obslužný mód – obsluha relevantních koncových zařízení
- funkce `void printUsage()` a `void printUsageInteraction()`:
  vypisují nápovědu a návod na použití aplikace
- funkce `void configureDevices()`: umožňuje konfiguraci všech relevantních koncových zařízení (nastavení sériového čísla jednoznačně identifikující koncová zařízení)
- funkce `void useDevices()`: slouží k obsluze všech relevantních koncových zařízení
- funkce `void userInteraction(USBDevice **usbDevices, int numberOfUSBDevices)`: čte příkazy uživatele a volá funkce, které provádí vstupně-výstupní operace s koncovými zařízeními
- funkce `void pollAndHandleEvents(USBDevice **usbDevices, int numberOfUSBDevices)`:
  zajišťuje obsluhu událostí, které byly poslány koncovými zařízeními
- funkce `void changemode(int dir)` a `int kbhit(void)`:
  mění nastavení standardního vstupu a umožňují čtení stisknuté klávesy

6.2.1 Pracovní módy aplikace

Po spuštění aplikace je zavolána hlavní funkce `main()`, která podle argumentů zadaných uživatelem pouze rozvětvuje chod programu na několik scénářů. V případě zvolení různých výpisů (nápověda, návod k použití, seznam zařízení atd.) se na standardní výstup vypíší požadované informace a program skončí. Dalšími možnostmi jsou dva módy, ve kterých může aplikace pracovat: konfigurační mód a obslužný mód.

Konfigurační mód

Tento mód umožňuje konfiguraci všech relevantních koncových zařízení. Můžeme přiřadit sériové čísla koncovým zařízením, která je jednoznačně identifikují. Hlavní funkce `main()` zavolá funkci `void configureDevices()`, která ve smyčce čte požadavky (příkazy) uživatele a podle nich nastavuje sériové číslo zvoleného koncového zařízení. Uživatel může kdykoliv ukončit program stisknutím klávesy `q`. Nejdříve je načten a vypsán seznam všech koncových zařízení s předdefinovanými hodnotami ID výrobce a ID produktu (v našem případě VID=0x04D8, PID=0x0204), jak naznačuje ukázka zdrojového kódu 12. K jednotlivým koncovým zařízením lze v tuto chvíli přistupovat pouze pomocí indexu. Zdrojový kód 13 ukazuje, jak je v seznamu hledáno koncové zařízení podle indexu zadaného uživatelem a následné nastavení nového ID sériového čísla.

```
// get information about all relevant USB devices
USBDeviceInfo usbDevicesInfos[MAX_NUMBER_OF_DEVICES];
int numberOfInfos = USBService::getUSBDeviceInfos(&usbDevicesInfos[0]);

// list relevant USB devices
USBService::listDevices(true, true);
```

Zdrojový kód 12: Načtení koncových zařízení v PC aplikaci

```
// loop through all devices in the list and find the one identified
// by the index
for (int i = 0; i < numberOfInfos; i++)
{
    if (usbDevicesInfos[i].index == (unsigned int)index)
    {
        unsigned short newSerialNumberId[SERIAL_NUMBER_ID_SIZE] = {
            (commandParts.at(0).compare("r") == 0 ? 'R' : 'U'),
            (unsigned short)commandParts.at(2).c_str()[0],
            (unsigned short)commandParts.at(3).c_str()[0]};

        USBService *usbService = USBService::instance();
        usbService->init();

        USBDevice usbDevice(usbService,
            usbDevicesInfos[i].serialNumberId);
        int r = usbDevice.connectUSBDevice(usbDevicesInfos[i].index);
        if (r == 0)
        {
            usbDevice.setDeviceId(newSerialNumberId);
            usbDevice.disconnectUSBDevice();
        }
        else
        {
            cout << "Cannot set the device id. Cannot connect the USB
device. " << r << endl;
        }

        usbService->close();
        USBService::destroy();

        break;
    }
}
```

Zdrojový kód 13: Nastavení ID koncového zařízení z PC aplikace

Obslužný mód

Aplikace bude zřejmě pracovat většinu času v obslužném módu, kdy program ovládá všechna nakonfigurovaná relevantní koncová zařízení. Hlavní funkce `main()` zavolá funkci `useDevices()`, která aktivuje knihovnu LibUSB, získá seznam všech nakonfigurovaných relevantních koncových zařízení a zavolá funkci `userInteraction()` (viz zdrojový kód 14). Ta nejprve připojí nalezená koncová zařízení a poté vstoupí do smyčky, ve které program provádí svou činnost dokud uživatel nestiskne klávesu, což je kontrolováno funkcí `kbhit()`.

Podle stisknuté klávesy se program rozhodne, kterou vstupně-výstupní operaci obsloužit a zavolá funkci některého z modulů. Poté co funkce `userInteraction()` skončí (uživatel stiskne klávesu `q`), je vráceno řízení funkci `useDevices()`, která uvolní seznam koncových zařízení, deaktivuje knihovnu LibUSB a program končí. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `main.cpp`.

```
// get information about all relevant USB devices
USBDeviceInfo usbDevicesInfos[MAX_NUMBER_OF_DEVICES];
int numberOfUSBDevices =
    USBService::getUSBDeviceInfos(&usbDevicesInfos[0], true);
...

// initialize the USB service
USBService *usbService = USBService::instance();
usbService->init();

// array of pointers to USBDevice objects
USBDevice *usbDevices[numberOfUSBDevices];
// populate the array of pointers to USBDevice objects
for (int i = 0; i < numberOfUSBDevices
    && i < (int)MAX_NUMBER_OF_DEVICES; i++)
{
    USBDevice *usbDevice = new USBDevice(usbService,
        usbDevicesInfos[i].serialNumberId);
    usbDevices[i] = usbDevice;
}

userInteraction(usbDevices, numberOfUSBDevices);

// disconnect all USB devices and destroy all USBDevice objects
for (int i = 0; i < numberOfUSBDevices
    && i < (int)MAX_NUMBER_OF_DEVICES; i++)
{
    usbDevices[i]->disconnectUSBDevice();
    delete usbDevices[i];
    usbDevices[i] = NULL;
}

// destroy the USB service
usbService->close();
USBService::destroy();
```

Zdrojový kód 14: Obslužný mód PC aplikace

6.2.2 USB modul

USB modul se skládá ze dvou tříd resp. čtyř souborů:

- `USBService.h`
- `USBService.cpp`
- `USBDevice.h`
- `USBDevice.cpp`

Hlavičkové soubory `USBService.h` a `USBDevice.h` obsahují deklarace atributů a metod a soubory `USBService.cpp` a `USBDevice.cpp` definice metod třídy `USBService` resp. `USBDevice`.

Třída USBService

Třída `USBService` je implementována pomocí vytvářecího návrhového vzoru Singleton. V aplikaci tedy může existovat pouze jedna instance této třídy, čímž máme zajištěn přístup ke knihovně LibUSB v rámci jediného kontextu. `USBService` obsahuje následující atributy:

- `static USBService* pinstance`: udržuje referenci na jedinou instanci třídy `USBService`
- `libusb_context *libusbContext`: udržuje referenci na kontext knihovny LibUSB

Seznam veřejných metod třídy `USBService` je následující:

- `static USBService * instance()`: vrací ukazatel na jedinou instanci třídy `USBService`, pokud ještě neexistuje, vytvoří novou
- `static void destroy()`: zruší jedinou instanci třídy `USBService`
- `void init()`: inicializuje knihovnu LibUSB
- `void close()`: uvolňuje knihovnu LibUSB
- `static void listDevices(bool relevant, bool forConfig)`: vypisuje seznam koncových zařízení. Parametr `relevant` určuje, zda se mají vypsat všechny, nebo jen relevantní zařízení a parametr `forConfig` určuje formát seznamu podle pracovního módu.
- `static int getUSBDeviceInfos(USBDeviceInfo *usbDeviceInfos, bool registered = false)`: slouží k získání základních informací o koncových zařízeních (index v seznamu a sériové číslo). Naplňuje pole struktur `USBDeviceInfo` a vrací počet těchto struktur.
- `int connectUSBDevice(libusb_device_handle **libusbDeviceHandle, const unsigned short *serialNumberId, ssize_t deviceNumber = -1)`: připojuje koncové zařízení ke knihovně LibUSB
- `void disconnectUSBDevice(libusb_device_handle *libusbDeviceHandle)`: odpojuje koncové zařízení od knihovny LibUSB
- `void handleEvents()`: obsluhuje čekající události v neblokujícím módu

Ukázka zdrojového kódu 15 obsahuje zkrácenou verzi metody `connectUSBDevice()` ze třídy `USBService`, bez kontroly chybových návratových hodnot volaných funkcí knihovny LibUSB. Metoda `connectUSBDevice()` připojuje koncové zařízení ke knihovně LibUSB. Nejdříve je získán seznam koncových zařízení (`libusb_get_device_list`), který je procházen ve smyčce `for`, kde se porovnává hodnota parametru `deviceNumber` s indexem koncového zařízení v seznamu. To je využíváno v konfiguračním módu, kdy koncová zařízení nemusí mít nastavené sériové číslo, takže se k nim musí přistupovat pomocí indexu, tedy hodnotou parametru `deviceNumber`, který je větší než 0. Potom je načten deskriptor zařízení (`libusb_get_device_descriptor`), podle něhož lze zkontrolovat ID výrobce a ID produktu. Dále je nutné v knihovně LibUSB „otevřít“ koncové zařízení, zkontrolovat jeho sériové číslo (viz zdrojový kód 16), nastavit aktivní konfiguraci a získat přístup k rozhraní, což umožňuje následné provádění vstupně-výstupních operací na koncových bodech. Nakonec je nutné uvolnit seznam koncových zařízení pomocí funkce `libusb_free_device_list()`. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `USBService.cpp`.

```
libusb_device **devs = NULL;    // pointer to pointer of device
int r;          // for return values
ssize_t cnt;    // holding number of devices in list

// get the list of devices
cnt = libusb_get_device_list(libusbContext, &devs);

ssize_t i;  // for iterating through the list
for (i = 0; i < cnt; i++)
{
    // check the device number (index in the list)
    if (deviceNumber > 0 && i != deviceNumber)
        continue;

    libusb_device *dev = devs[i];

    libusb_device_descriptor desc;
    r = libusb_get_device_descriptor(dev, &desc);

    // check if the USB device has correct/relevant
    // vendor id and and product id
    if (desc.idVendor == MY_VID && desc.idProduct == MY_PID)
    {
        // opens a USB device
        r = libusb_open(dev, libusbDeviceHandle);

        // CHECK IF THE USB DEVICE HAS CORRECT SERIAL NUMBER ID

        // sets the active configuration of the device
        r = libusb_set_configuration(*libusbDeviceHandle, 1);

        // claims the interface with the Operating System
        r = libusb_claim_interface(*libusbDeviceHandle, 0);

        cout << "connection successful" << endl;
        break;
    }
}

// free the list, unref the devices in it
libusb_free_device_list(devs, 1);
```

Zdrojový kód 15: Připojení USB koncového zařízení z třídy USBService

```
// check if the USB device has correct serial number id
int numberOfReturnedData = 0;
int dataSizeString = 100;
unsigned char dataArray[dataSizeString];
bool serialNumberIdMatch = true;

numberOfReturnedData =
    libusb_get_string_descriptor_ascii(*libusbDeviceHandle,
        desc.iSerialNumber, dataArray, dataSizeString);
for (int i = 0; i < dataSizeString && i < numberOfReturnedData; i++)
{
    // if the first part of the serial number does not match
    // the default serial number  or the serial number id
    // does not match, check another USB device
    if ((i < (int)SERIAL_NUMBER_ID_INDEX
        && DEFAULT_SERIAL_NUMBER[i] != dataArray[i])
        || (i >= (int)SERIAL_NUMBER_ID_INDEX
        && serialNumberId[i - SERIAL_NUMBER_ID_INDEX] != dataArray[i]))
    {
        serialNumberIdMatch = false;
        break;
    }
}

if (!serialNumberIdMatch)
{
    // void libusb_close(libusb_device_handle *dev_handle)
    libusb_close(*libusbDeviceHandle);
    *libusbDeviceHandle = NULL;
    continue;
}
```

Zdrojový kód 16: Kontrola sériového čísla koncového zařízení

Třída USBDevice

Třída `USBDevice` reprezentuje rozhraní USB koncového zařízení, přes které jsou prováděny veškeré vstupně-výstupní operace. `USBDevice` si udržuje referenci na třídu `USBService`, pomocí níž připojuje a odpojuje koncová zařízení. Dále obsahuje informace o stavu připojení, sériové číslo koncového zařízení, ukazatel na LibUSB ovladač USB koncového zařízení (`libusb_device_handle`) a reference na synchronní a asynchronní komunikační moduly. Díky tomu je hlavní modul odstíněn od způsobu komunikace přes USB a od samotné knihovny LibUSB, která tak může být nahrazena jinou knihovnou umožňující přístup ke koncovým zařízením. Třída obsahuje také metody tvořící v podstatě seznam všech možných vstupně-výstupních operací, které mohou být prováděny na koncovém zařízení. Ukázka zdrojového kódu 17 obsahuje metodu `connectUSBDevice()` ze třídy `USBDevice`, která připojuje koncové zařízení ke knihovně LibUSB tím, že zavolá metodu `connectUSBDevice()` ze třídy `USBService` (viz zdrojový kód 15). Na ukázce zdrojového kódu 18 pak vidíme jednoduché převolání metody `getTemperature()` ve třídě `USBCommunicationSync` v modulu synchronní komunikace. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `USBDevice.cpp`.

```
int USBDevice::connectUSBDevice(ssize_t deviceNumber)
{
    if (connectionStatus == CONNECTED)
        return 0;

    int result = usbService->connectUSBDevice(&libusbDeviceHandle,
        serialNumberId, deviceNumber);

    // if everything went well, set connection status to CONNECTED
    if (result == 0)
    {
        usbCommunicationSync.setLibusbDeviceHandle(libusbDeviceHandle);
        connectionStatus = CONNECTED;
    }

    return result;
}
```

Zdrojový kód 17: Připojení USB koncového zařízení z třídy USBDevice

```
void USBDevice::getTemperature()
{
    if (connectionStatus == CONNECTED)
        usbCommunicationSync.getTemperature();
}
```

Zdrojový kód 18: Získání teploty z USB koncového zařízení z třídy USBDevice

6.2.3 Modul synchronní komunikace přes USB

Do tohoto modulu patří pouze dva soubory:

- `USBCommunicationSync.h`
- `USBCommunicationSync.cpp`

Hlavičkový soubor `USBCommunicationSync.h` obsahuje deklarace atributů a metod a soubor `USBCommunicationSync.cpp` definice metod třídy `USBCommunicationSync`, která si udržuje ukazatel na LibUSB ovladač USB koncového zařízení (`libusb_device_handle`) a obsahuje dvě pole reprezentující vyrovnávací paměti koncových bodů pro vstup a výstup (angl. endpoint in/out buffers). Následující třídní metody tvoří výčet všech možných vstupně-výstupních operací, které mohou být prováděny na koncovém zařízení při použití synchronní komunikace:

- `int setDeviceId(const unsigned short *serialNumberId)`: nastavuje nové ID koncového zařízení (část sériového čísla)
- `void getDeviceId()`: vypisuje ID koncového zařízení
- `void getData()`: vypisuje data programové paměti mikrokontroléru z předdefinované adresy
- `void toggleLED(unsigned char command)`: rozsvěcuje resp. zhasíná LED. Parametr `command` určuje, která (které) LED se má (mají) rozsvítit resp. zhasnout.
- `void getTemperature()`: vypisuje teplotu naměřenou koncovým zařízením
- `void sendDate()`: posílá aktuální datum na koncové zařízení

Ukázka zdrojového kódu 19 obsahuje metodu `getTemperature()` ze třídy `USBCommunicationSync`, která získává teplotu z koncové zařízení. Dochází zde ke dvěma voláním funkce `libusb_bulk_transfer` knihovny LibUSB, nejdříve s parametry odchozího koncového bodu (`EP_OUT`) a následně s parametry příchozího koncového bodu (`EP_IN`). Jedná se tedy o komunikaci směrem od kořenového uzlu ke koncovému zařízení. Pokud přehodíme volání funkcí (nejdříve s parametry odchozího, a potom příchozího koncového bodu) získáme komunikaci směrem od koncového zařízení ke kořenovému uzlu. Oba případy zahrnují jak dotaz tak odpověď. Můžeme však použít i jednostrannou komunikaci, kdy voláme funkci `libusb_bulk_transfer` pouze s parametry odchozího koncového bodu (`EP_OUT`): např. nastavení sériového čísla koncového zařízení nebo změna stavu LED. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `USBCommunicationSync.cpp`.

```
void USBCommunicationSync::getTemperature()
{
    int result;
    float temperature;
    int transferredValue = 0;
    int *transferred = &transferredValue;

    // set the "Get Temperature" command (described in the firmware)
    OutputPacketBuffer[0] = COMMAND_GET_TEMPERATURE;

    // send the command - BUFFER_LENGTH bytes of data to the USB device
    int errorCode = libusb_bulk_transfer(libusbDeviceHandle, EP_OUT,
        &OutputPacketBuffer[0], BUFFER_LENGTH, transferred, 5000);

    if (*transferred != BUFFER_LENGTH)
    {
        cerr << "Temerature: Write Failed with error code "
            << errorCode << ", transferred " << *transferred
            << " bytes." << endl;
        return;
    }

    // now get the response packet from the firmware - retrieves
    // BUFFER_LENGTH bytes of data from the USB device
    errorCode = libusb_bulk_transfer(libusbDeviceHandle, EP_IN,
        &InputPacketBuffer[0], BUFFER_LENGTH, transferred, 5000);

    if (*transferred != BUFFER_LENGTH)
    {
        cerr << "Temperature: Read Failed with error code "
            << errorCode << ", transferred " << *transferred
            << " bytes." << endl;
        return;
    }

    // calculate and print out the temperature
    // InputPacketBuffer[0] is an echo back of the command
    // InputPacketBuffer[1] contains the MSB value
    // of the temperature register
    // InputPacketBuffer[2] contains the LSB value
    // of the temperature register
}
```

Zdrojový kód 19: Získání teploty

6.2.4 Modul asynchronní komunikace přes USB

Do tohoto modulu patří pouze dva soubory:

- `USBCommunicationAsync.h`
- `USBCommunicationAsync.cpp`

Hlavičkový soubor `USBCommunicationAsync.h` obsahuje deklarace atributů a metod a soubor `USBCommunicationAsync.cpp` definice metod třídy `USBCommunicationAsync`, která si udržuje informace o stavu asynchronní komunikace, zda je aktivováno automatické čtení teploty a zda jsou přijímány události z koncového zařízení. Obsahuje tři pole reprezentující vyrovnávací paměti koncových bodů pro vstup a výstup (angl. endpoint in/out buffers), kde jedno z nich je vyhrazeno pro přenos událostí. Dále pak tři struktury přenosu dat (`libusb_transfer`). Na rozdíl od třídy `USBCommunicationSync` určené pro synchronní komunikaci si třída `USBCommunicationAsync` neudržuje ukazatel na LibUSB ovladač USB koncového zařízení (`libusb_device_handle`). Ten je získáván při alokaci paměti pro přenos dat metodou:

```
int allocateTransfers(libusb_device_handle *libusbDeviceHandle)
```

Následující třídní metody tvoří výčet všech možných vstupně-výstupních operací, které mohou být prováděny na koncovém zařízení při použití asynchronní komunikace:

- `int allocateTransfers(libusb_device_handle *libusbDeviceHandle)`: alokuje paměť pro přenos dat
- `void deallocateTransfers()`: dealokuje paměť pro přenos dat
- `void activateTemperatureReading()`: aktivuje automatické čtení teploty
- `void deactivateTemperatureReading()`: deaktivuje automatické čtení teploty
- `void activateEventHandler()`: aktivuje příjem událostí z koncového zařízení
- `void deactivateEventHandler()`: deaktivuje příjem událostí z koncového zařízení
- `int submitTemperatureReadingTransfers()`: posílá požadavek na získání teploty
- `int submitEventTransfer()`: posílá požadavek na přenos události

Kromě dalších metod obsahuje třída chráněné metody reprezentující funkce zpětného volání:

- `static void callbackCommandOut(struct libusb_transfer *transfer)`
- `static void callbackResponseIn(struct libusb_transfer *transfer)`
- `static void callbackEventIn(struct libusb_transfer *transfer)`

Každá úloha, kterou třída `USBCommunicationAsync` řeší, je rozdělená do několika funkcí, což odpovídá principu asynchronního přenosu dat implementovaného knihovnou LibUSB popsaného v kapitole 3.5.5. Ukázky zdrojových kódů 20 až 24 implementují scénář automatického získávání teploty a částečně příjem událostí z koncového zařízení. Zdrojový kód 20 popisuje alokaci a naplnění všech datových přenosů:

- pro čtení teploty: odchozí = `transferCommandOut` a příchozí = `transferResponseIn`
- pro příjem událostí z koncového zařízení: pouze přízhozí = `transferEventIn`.

Na zdrojovém kódu 21 vidíme aktivaci automatického čtení teploty z koncového zařízení. Volá se zde funkce `submitTemperatureReadingTransfers()`, která posílá požadavky na přenos dat pomocí funkce knihovny LibUSB: `libusb_submit_transfer` (viz zdrojový kód 22). Po každém přenosu dat je zavolána funkce zpětného volání (viz podkapitola „Obsluha událostí“). V případě odchozího přenosu dat stačí pouze vypsat informaci o neúspěchu (viz zdrojový kód 23), ale ve funkci zpětného volání příchozího přenosu dat získáváme informace zaslané koncovým zařízením, tedy odpověď, kterou je třeba zpracovat, jak naznačuje zdrojový kód 24. Více informací je možno získat přímo ze zdrojových kódů programu pro mikrokontrolér: příloha F, soubor `USBCommunicationAsync.cpp`.

```
int USBCommunicationAsync::allocateTransfers(
    libusb_device_handle *libusbDeviceHandle)
{
    // Define transfer of data IN (IN to host PC from USB-device)
    transferResponseIn = libusb_alloc_transfer(0);

    // populate the required libusb_transfer fields for a bulk
    // transfer, use this object as the user_data parameter,
    // timeout - value of 0 indicates no timeout
    // (see struct libusb_transfer).
    libusb_fill_bulk_transfer(transferResponseIn,
        libusbDeviceHandle, EPA_IN,
        asyncInputPacketBuffer, BUFFER_LENGTH,
        callbackResponseIn, this, 0);

    // Define transfer of data OUT (OUT to USB-device from host PC)
    transferCommandOut = libusb_alloc_transfer(0);

    libusb_fill_bulk_transfer(transferCommandOut,
        libusbDeviceHandle, EPA_OUT,
        asyncOutputPacketBuffer, BUFFER_LENGTH,
        callbackCommandOut, this, 0);

    // Define transfer of data IN (IN to host PC from USB-device)
    transferEventIn = libusb_alloc_transfer(0);

    libusb_fill_bulk_transfer(transferEventIn,
        libusbDeviceHandle, EPE_IN,
        asyncEventInputPacketBuffer, BUFFER_LENGTH,
        callbackEventIn, this, 0);
    return 0;
}
```

Zdrojový kód 20: Alokace asynchronních datových přenosů

```
void USBCommunicationAsync::activateTemperatureReading()
{
    // set the "Get Temperature Value" command to the buffer
    asyncOutputPacketBuffer[0] = COMMAND_ASYNC_GET_TEMPERATURE;
    int r = submitTemperatureReadingTransfers();
    autoTemperatureReading = true;
}
```

Zdrojový kód 21: Aktivace automatického čtení teploty

```
int USBCommunicationAsync::submitTemperatureReadingTransfers()
{
    int r;

    // if one of the transfer pointers is NULL,
    // then return - do not submit transfers
    if (!transferResponseIn || !transferCommandOut)
        return 1;

    // if the temperature reading data transfer is pending,
    // then return - do not submit transfers
    if (temperatureReadingPending)
        return 2;

    // disable resubmition of automatic temperature reading transfers
    temperatureReadingPending = true;

    // submit transfer of data IN (IN to host PC from USB-device)
    r = libusb_submit_transfer(transferResponseIn);

    // submit transfer of data OUT (OUT to USB-device from host PC)
    r = libusb_submit_transfer(transferCommandOut);

    return 0;
}
```

Zdrojový kód 22: Odeslání požadavku na získání teploty z koncového zařízení

```
void USBCommunicationAsync::callbackCommandOut(
    struct libusb_transfer *transfer)
{
    // check the transfer status
    if (transfer->status != 0
            || transfer->actual_length < BUFFER_LENGTH)
        cerr << "Error: callbackCommandOut: status = "
            << transfer->status << ", actual_length = "
            << transfer->actual_length << endl;
}
```

Zdrojový kód 23: Kontrola odeslání požadavku na čtení teploty ve funkci zpětného volání (angl. callback function)

```
void USBCommunicationAsync::callbackResponseIn(
    struct libusb_transfer *transfer)
{
    // get the sender object stored in the user_data
    USBCommunicationAsync *sender =
        (USBCommunicationAsync *)transfer->user_data;

    if (!sender->isAutoTemperatureReading())
        return;

    // check the transfer status and actual length
    // of the transfer buffer
    if (transfer->status != 0
            || transfer->actual_length != BUFFER_LENGTH)
        cerr << "Error: callbackResponseIn: status = "
            << transfer->status << ", actual_length = "
            << transfer->actual_length << endl;

    // calculate and print out the temperature

    // enable the resubmition of automatic temperature reading
    // transfers
    sender->setTemperatureReadingPending(false);
}
```

Zdrojový kód 24: Čtení teploty ve funkci zpětného volání (angl. callback function)

Obsluha událostí

Ukázka zdrojového kódu 25 naznačuje, jakým způsobem jsou obsluhovány události. V hlavní smyčce je neustále kontrolováno, zda uživatel stiskl nějakou klávesu, jinak se volá funkce obsluhy událostí (viz zdrojový kód 26). Pro každé koncové zařízení se nejdříve zavolá funkce synchronního modulu: `sendDate()`, která kontroluje, jestli koncové zařízení chtělo získat aktuální datum z PC aplikace. Pokud je aktivováno automatické čtení teploty resp. příjem událostí (alarmu), jsou volány funkce `submitTemperatureReadingTransfers()` resp. `submitEventTransfer()` z asynchronního modulu, které posílají požadavky na přenos dat. Nakonec je pouze jednou (už ne pro každé zařízení zvlášť) zavolána funkce `handleEvents()` ze třídy `USBService` (viz zdrojový kód 27), která obslouží asynchronní události, tedy zajistí volání funkcí zpětného volání (angl. callback functions). Třída `USBCommunicationAsync` si navíc uchovává informaci o stavech přenosů dat, takže při vícenásobném volání funkcí `submitTemperatureReadingTransfers()` resp. `submitEventTransfer()` se kontroluje, zda právě neprobíhá přenos dat. Pokud ne, může být poslán nový požadavek na přenos dat.

```
while (actionKey != 'q')
{
    // do something until a key is pressed
    while (!kbhit())
    {
        pollAndHandleEvents(usbDevices, numberOfUSBDevices);
    }
    // get the key pressed
    actionKey = cin.get();
    ...
}
```

Zdrojový kód 25: Volání funkce pro obsluhu událostí z hlavní smyčky

```
void pollAndHandleEvents(USBDevice **usbDevices, int
numberOfUSBDevices)
{
    for (int i = 0; i < numberOfUSBDevices; i++)
    {
        // poll the USB device synchronous requests

        // check if the USB device requested to get the current date
        usbDevices[i]->sendDate();

        // submit the USB device asynchronous data transfers

        // if the automatic temperature reading is active,
        // submit transfers again
        if (usbDevices[i]->isAutoTemperatureReading())
        {
            // submit transfers again
            int r = usbDevices[i]->submitTemperatureReadingTransfers();
            if (r != 0 && r != 2)
                cerr << "Error: Failed to resubmit automatic
temperature reading transfers. " << r << endl;
        }

        // if the event handling is active, submit transfer again
        if (usbDevices[i]->isAcceptingEvents())
        {
            // submit transfers again
            int r = usbDevices[i]->submitEventTransfer();
            if (r != 0 && r != 1)
                cerr << "Error: Failed to resubmit event transfer. "
                    << r << endl;
        }
    }

    // handle events
    USBService::instance()->handleEvents();
}
```

Zdrojový kód 26: Funkce obsluhy událostí

```
void USBService::handleEvents()
{
    const libusb_pollfd **libusbFileDescriptors =
        libusb_get_pollfds(libusbContext);

    nfds_t numberOfLibusbFileDescriptors = 0;
    while (libusbFileDescriptors[numberOfLibusbFileDescriptors]!=NULL)
        numberOfLibusbFileDescriptors++;

    if (numberOfLibusbFileDescriptors > 0)
    {
        struct pollfd
            pollFileDescriptors[numberOfLibusbFileDescriptors];
        for (int i = 0; i < (int)numberOfLibusbFileDescriptors; i++)
        {
            pollFileDescriptors[i].fd = libusbFileDescriptors[i]->fd;
            pollFileDescriptors[i].events =
                libusbFileDescriptors[i]->events;
        }

        //int poll(struct pollfd fds[], nfds_t nfds, 0);
        // check if poll() indicated activity
        // on libusb file descriptors
        if (poll(pollFileDescriptors, numberOfLibusbFileDescriptors, 0)
                > 0)
            libusb_handle_events_timeout(libusbContext, 0);
    }
}
```

Zdrojový kód 27: Obsluha událostí a volání funkcí zpětného volání (angl. callback functions)

6.2.5 Modul podpůrných nástrojů

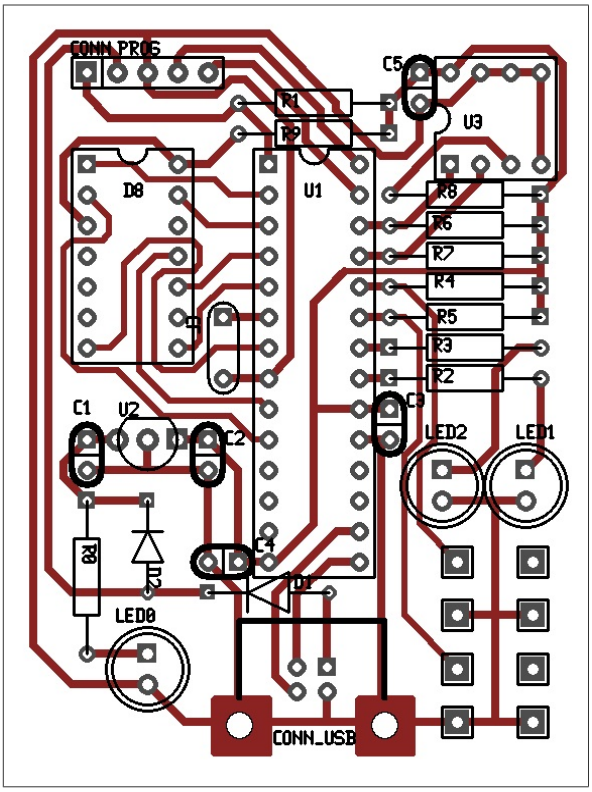
Poslední modul obsahuje následující soubory:

- `utils.h`
- `utils.cpp`
- `Timer.h`
- `Timer.cpp`
- `Logger.h`
- `Logger.cpp`

Hlavičkový soubor `utils.h` obsahuje deklarace funkcí a soubor `utils.cpp` definice funkcí, které jsou užitečné při práci s řetězci znaků. Hlavičkové soubory `Timer.h` a `Logger.h` obsahují deklarace atributů a metod a soubory `Timer.cpp` a `Logger.cpp` definice metod třídy `Timer` resp. `Logger`. Třída `Timer` slouží k měření času (viz kapitola 7.4) a třída `Logger` slouží k zaznamenávání informačních zpráv pro testování a ladění aplikace (viz kapitola 7.5.6).

7 Ověření funkcionality

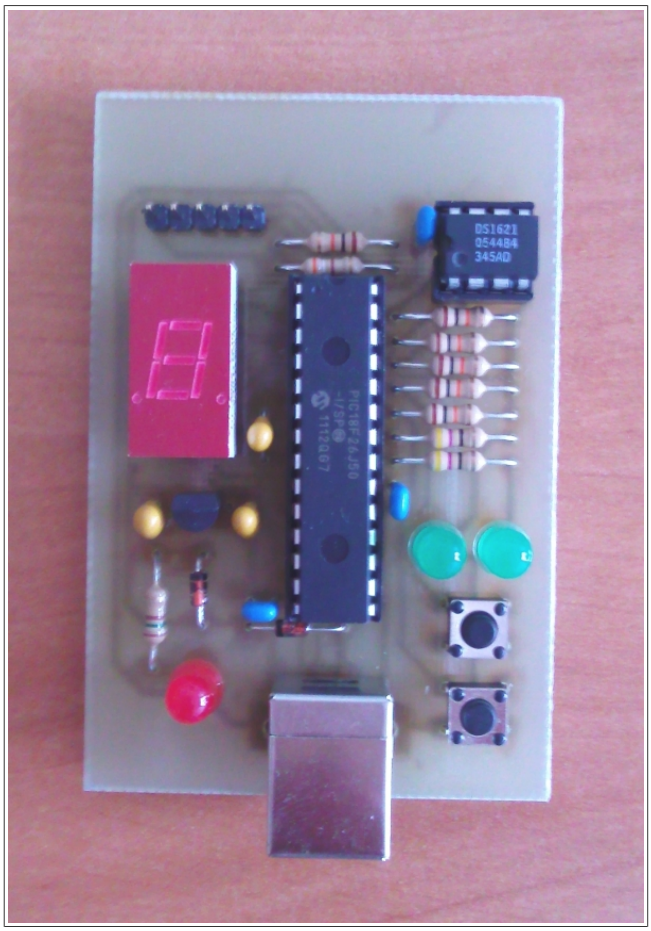
V této fázi bychom měli mít vytvořené dvě aplikace: firmware mikrokontroléru PIC18F26J50 a PC aplikaci. Na nepájivém poli máme zapojení naprogramovaného mikrokontroléru s periferiemi. Nyní můžeme vyzkoušet, jestli PC aplikace dokáže komunikovat s jedním koncovým zařízením. Pro ověření komunikace s více koncovými zařízeními musíme zapojit více mikrokontrolérů se svými vlastními periferiemi, tedy sestavit několik zařízení. K tomuto účelu jsem podle schématu zapojení na obrázku 11 navrhl desku plošných spojů (viz obrázek 18). V další kapitole 7.1 je stručně popsán celý proces výroby desky plošných spojů.



Ilustrace 18: Deska plošných spojů

7.1 Výroba desky plošných spojů

Stručný popis výroby desky plošných spojů, který jsem použil, je následující. Nejprve je nutné navrhnout desku plošných spojů pomocí nějakého softwaru (např. PCB, EAGLE atd.). Já jsem použil program PCB, který je součástí volně šiřitelného balíku gEDA. Navrženou desku vytiskneme na průhlednou fólii. Zakoupíme si fotosenzitivní jednostranný plošný spoj (angl. fotocuprextit), ze kterého vyřízneme část odpovídající navržené desce plošných spojů. Na fotosenzitivní vrstvu desky přiložíme připravenou průhlednou fólii a osvítíme ultrafialovým světlem. Ale pozor, je důležité, aby potištěná strana fólie byla přiložena na desce. Jinak by mohlo dojít k nežádoucímu prosvícení fólie, která má také jistou tloušťku, a spoje na desce by po vyleptání mohly být příliš tenké. Potom desku vyvoláme pomocí vývojky a vyleptáme v leptacím roztoku. Nakonec vyvrtáme v desce díry, osadíme ji součástkami, které připájíme k pájecím bodům. Podrobný popis výroby desky plošných spojů lze najít v mnoha odborných literaturách nebo na internetu (např. literatura [17]). Výsledné USB koncové zařízení lze vidět na obrázku 19.



Obrázek 19: USB koncové zařízení

7.2 Konfigurace koncových zařízení

Před tím, něž USB koncová zařízení můžeme ovládat, musíme je nakonfigurovat tak, aby byla jednoznačně identifikovatelná. K tomu slouží konfigurační mód PC aplikace, do kterého se dostaneme spuštěním PC aplikace pomocí následujícího příkazu:

```
./USBThermometer -c
```

Pokud není k počítači připojeno žádné relevantní koncové zařízení, tedy demonstrační zařízení, které je náplní této diplomové práce, PC aplikace vypíše následující návod k obsluze s počtem všech USB koncových zařízení, které jsou připojeny k počítači, kdy máme pouze možnost ukončit aplikaci:

```
CONFIGURATION MODE
------------------

It is possible to type one of the following commands to perform
an action:
r,I,A,B    Register the USB device (set the id to RAB),
u,I,A,B    Unregister the USB device (set the id to UAB),
d,I        Unregister the USB device (set the id to default XYZ),
where r/u/d specifies actual command, I specifies the device index,
and A/B could be any characters used as the device ID (serial number
part).
4 USB devices connected to the PC.

-------------------------------------------------------------------------
Index IDVendor    IDProduct   SerialNumber      Type
-------------------------------------------------------------------------
Enter the command or q to quit:
```

V případě, že k počítači je připojeno alespoň jedno relevantní koncové zařízení (demonstrační zařízení), PC aplikace vypíše následující návod k obsluze s počtem všech USB koncových zařízení, které jsou připojeny k počítači a čeká na příkaz zadaný uživatelem:

```
CONFIGURATION MODE
------------------

It is possible to type one of the following commands to perform
an action:
r,I,A,B    Register the USB device (set the id to RAB),
u,I,A,B    Unregister the USB device (set the id to UAB),
d,I        Unregister the USB device (set the id to default XYZ),
where r/u/d specifies actual command, I specifies the device index,
and A/B could be any characters used as the device ID (serial number
```

```
part).
5 USB devices connected to the PC.

------------------------------------------------------------------------
Index IDVendor    IDProduct   SerialNumber      Type
------------------------------------------------------------------------
4     4d8         204         'T01-SA-R-C'      registered
Enter the command or q to quit:
```

Uživatel má na výběr ze tří následujících příkazů:

- `r,I,A,B`: zaregistrování koncového zařízení, např. příkaz `r,0,x,y` nastaví sériové číslo koncového zařízení na hodnotu `T01-SA-Rxy`
- `u,I,A,B`: odregistrování koncového zařízení, např. příkaz `u,0,x,y` nastaví sériové číslo koncového zařízení na hodnotu `T01-SA-Uxy`
- `d,I`: nastavení sériového čísla koncového zařízení na výchozí hodnotu, např. příkaz `d,0` nastaví sériové číslo koncového zařízení na hodnotu `T01-SA-XYZ`

Význam sériového čísla je následující. První písmeno T označuje druh zařízení, v tomto případě teploměr (angl. thermometer), číslo 01 udává typ/verzi zařízení, písmena SA označují typ použité komunikace (S = synchronní, A = asynchronní) a poslední 3 znaky jsou vyhrazeny pro jednoznačnou identifikaci koncového zařízení, přičemž hodnota XYZ je výchozí, hodnota začínající písmenem U označuje neregistrované zařízení (může být nastaveno PC aplikcí v případě, že k počítači resp. k PC aplikaci budou připojeny koncová zařízení, která mají nakonfigurovaná shodná sériová čísla), a hodnota začínající písmenem R označuje registrované koncové zařízení, se kterým může (je ochotna) PC aplikace komunikovat. Např. sériové číslo s hodnotou:

```
T01-SA-Rxy
```

označuje koncové zařízení: teploměr, první svého druhu, které dokáže komunikovat s počítačem pomocí synchronního i asynchronního rozhraní a je registrováno, takže s ním PC aplikace může komunikovat.

7.3 Obsluha koncových zařízení z PC aplikace

PC aplikace může ovládat pouze registrovaná USB koncová zařízení. K tomu slouží obslužný mód, do kterého se dostaneme spuštěním PC aplikace pomocí následujícího příkazu:

```
./USBThermometer
```

Pokud není k počítači připojeno žádné relevantní registrované koncové zařízení, PC aplikace vypíše následující informaci o tom, že nemůže najít žádné zařízení:

```
Cannot find any registered USB device. Make sure there is a USB device
connected to the PC and/or it is registered (use --configure parameter
to configure USB devices).
```

Potom čeká maximálně 10 sekund, dokud není připojeno žádné zařízení (vypíše jednu tečku za sekundu – maximálně 10 teček):

```
..........
```

Pře ukončením znovu vypíše informaci o tom, že nemůže najít žádné zařízení:

```
Cannot find any registered USB device. Make sure there is a USB device
connected to the PC and/or it is registered (use --configure parameter
to configure USB devices).
```

V případě, že k počítači je připojeno alespoň jedno registrované koncové zařízení, PC aplikace vypíše následující: počet všech USB koncových zařízení, které jsou připojeny k počítači, informace o úspěšně připojených koncových zařízeních (identifikátory `R-C` a `R-A`), návod k obsluze. A čeká na příkaz zadaný uživatelem:

```
6 USB devices in list.
connection successful
USB device R-C has been successfully connected.
DEBUG main.userInteraction(): index=1
6 USB devices in list.
connection successful
USB device R-A has been successfully connected.
It is possible to press one of the following keys to perform an action:
h          Help - print this information again
g          Get the id
c          Get the data/content from the program memory
t          Read the actual temperature
a          Activate the automatic temperature reading
d          Deactivate the automatic temperature reading
q          Disconnect all devices and quit the application
<0-9>      A number 0-9 toggles LED(s), number means the following:
             0 - disables LED to be controled by this application,
             1/2/3 - turns on / turns off / toggles LED 1,
             4/5/6 - turns on / turns off / toggles LED 2,
             7/8/9 - turns on / turns off / toggles both LEDs.
Enter the command (h, q) or
Enter the USB device index followed by command (g, c, t, a, d, <0-9>):
Available USB device indexes: [0-1]
```

Uživatel stiskne klávesu – číslo koncového zařízení (podle předchozího výpisu má na výběr čísla 0 nebo 1) a potom stiskne klávesu reprezentující jeden z následujících příkazů:

- `h`: vypíše nápovědu
- `g`: vypíše identifikátor koncového zařízení, např. po sledu stisknutých kláves `0` a `g` vypíše ID zařízení 0 (`R-C`) a v závorkách hexadecimální podobu ID (každý znak má dva byty a nejnižší byte je vypsán jako první): `USB device ID: R-C  ( 52 0 2d 0 43 0 )`
- `c`: vypíše 15 bytů programové paměti mikrokontroléru z předdefinované adresy (0x002000), např. po sledu stisknutých kláves `1` a `c` vypíše 15 bytů ze zařízení 1 (`R-A`): `USB device data: |16|3|54|0|30|0|31|0|2d|0|53|0|41|0|2d|`
- `t`: vypíše teplotu naměřenou koncovým zařízením, např. po sledu stisknutých kláves `0` a `t` vypíše teplotu 25 °C naměřenou zařízením 0 (`R-C`): `Temperature: 25 °C`
- `a`: aktivuje automatické čtení teploty z koncového zařízení, např. po sledu stisknutých kláves `0` a `a` aktivuje automatické čtení teploty z koncového zařízení 0 (`R-C`), vypíše: `Automatic temperature reading has been activated.` a následně bude vypisovat: `--- Device R-C has sent the Async Temperature: 25 °C ---`
- `d`: deaktivuje automatické čtení teploty z koncového zařízení, např. po sledu stisknutých kláves `0` a `d` deaktivuje automatické čtení teploty z koncového zařízení 0 (`R-C`) a vypíše: `Automatic temperature reading has been deactivated.`
- `q`: odpojí všechna koncová zařízení a ukončí aplikaci
- `<0-9>` – tedy klávesa 0 – 9: rozsvítí nebo zhasne jednu nebo obě zelené LED, např. po sledu stisknutých kláves `1` a `7` rozsvítí obě zelené LED na zařízení 1 (`R-A`) a vypíše: `Toggle LED command has been sent.`

Výjimku tvoří příkazy (stisknuté klávesy) `h` a `q`, kterým nemusí předcházet stisknutá klávesa reprezentující číslo koncového zařízení.

Navíc může aplikace reagovat na dva typy událostí vyslané koncovým zařízením:

- získání data z PC aplikace: pokud stiskneme mikrospínač S2 (na obrázku 18 vrchní mikrospínač) na některém zařízení, PC aplikace pošle datum na toto zařízení a vypíše: `The USB device requested to get a current date. It was successfuly sent.`
- poslání události typu alarm: pokud stiskneme mikrospínač S1 (na obrázku 18 spodní mikrospínač) na některém zařízení, PC aplikace získá událost z tohoto zařízení a vypíše např. v případě zařízení 1 (`R-A`):

  `--- Device R-A has sent the ASYNC EVENT !!!`

7.4 Test rychlosti přenosu dat

Provedl jsem jednoduché testy rychlosti přenosu dat resp. odezvy PC aplikace při přenosu dat. Změřil jsem rychlost v PC aplikaci pomocí rozdílu času mezi odstartováním a ukončením přenosu dat. V případě synchronního rozhraní si před voláním funkce knihovny LibUSB uložíme startovní čas a po ukončení knihovní funkce si uložíme koncový čas. V programu může měření rychlosti vypadat, jako v ukázce zdrojového kódu 28.

```
timer.start();
int errorCode = libusb_bulk_transfer(libusbDeviceHandle, EP_OUT,
      &OutputPacketBuffer[0], BUFFER_LENGTH, transferred, 5000);
timer.stop();
```

Zdrojový kód 28: Měření rychlosti synchronního přenosu dat

U asynchronního rozhraní je situace trochu složitější. Startovní čas si uložíme v době odeslání požadavku na přenos dat a koncový čas ihned při začátku obsluhy události ve funkci zpětného volání (angl. callback function). Tento případ je naznačen v ukázce zdrojového kódu 29.

```
int USBCommunicationAsync::submitTemperatureReadingTransfers()
{
    int r;
    ...
    timerCommandOut.start();
    r = libusb_submit_transfer(transferCommandOut);
    ...
}

void USBCommunicationAsync::callbackCommandOut(
      struct libusb_transfer *transfer)
{
    // get the sender object stored in the user_data
    USBCommunicationAsync *sender =
        (USBCommunicationAsync *)transfer->user_data;

    sender->getTimerCommandOut()->stop();
    ...
}
```

Zdrojový kód 29: Měření rychlosti asynchronního přenosu dat

7.4.1 Měření času

Měření jsem provedl nejdříve pouze s jedním koncovým zařízením zapojeným na nepájivém poli, poté pouze s jedním koncovým zařízením osazeném na desce plošných spojů a nakonec jsem změřil rychlosti obou zařízení připojených k počítači a komunikujících s PC aplikací současně. Výsledky jsou obsaženy v příloze C v tabulkách 10 a 11 a v příloze D v tabulkách 12 a 13.

Jedno koncového zařízení komunikující s PC aplikací

Z tabulek 10 a 11 můžeme vyčíst, že průměrné doby trvání synchronního přenosu dat se pohybují v rozmezí 770 – 953 μs. V případě asynchronního přenosu dat je situace trochu lepší. Doba trvání asynchronního přenosu dat se pohybuje mezi 691 a 780 μs. Výjimku však tvoří průměrná hodnota 1 399 μs, kterou jsem zaznamenal u příchozího přenosu dat (od koncového zařízení k PC aplikaci) při automatickém čtení teploty z koncového zařízení na desce plošných spojů. V některých případech je doba přenosu dat větší o 1 ms. Nepodařilo se mi však zjistit důvod tohoto zpoždění.

Dvě koncová zařízení komunikující s PC aplikací současně

Z tabulek 12 a 13 můžeme vyčíst, že doba trvání synchronního i asynchronního přenosu dat, kdy s PC aplikací komunikují obě zařízení, je téměř stejná, jako při samostatně zapojených zařízeních. Stejně jako v předchozím případě vzniká při asynchronním příchozím přenosu dat (od koncového zařízení k PC aplikaci) při automatickém čtení teploty z koncového zařízení na desce plošných spojů zpoždění zhruba 1 ms.

Asynchronní přenos dat tedy dosahuje nepatrně lepších výsledků než synchronní. Rozdíly mezi zařízením zapojeným na nepájivém poli a zařízením na desce plošných spojů jsou zanedbatelné, až na výše popsané zpoždění 1 ms u zařízení na desce plošných spojů.

7.5 Test stability

Po připojení koncového zařízení k počítači se rozsvítí červená LED indikující přivedené napájecí napětí. Zároveň svítí zelená LED1 (pravá LED na obrázku 18) a sedmisegmentový displej zobrazuje výchozí datum, resp. formát data: DD-MM-YYYY---. Stisknutím tlačítka 1 nebo 2 se nic neděje. Následující krátké testy ukazují, jak se koncová zařízení chovají v různých situacích.

7.5.1 Zapnutí počítače

Tabulka 6 shrnuje chování koncového zařízení během zapínání počítače, startování operačního systému Linux a následném spuštění PC aplikace.

Operace	Chování koncového zařízení (změny)
Vypnuté PC	- svítí zelená LED1 (pravá LED na obrázku 18) - sedmisegmentový displej zobrazuje výchozí datum (DD-MM-YYYY---)
Zapnutí PC	- svítí zelená LED2 (levá LED na obrázku 18)
Startuje BIOS	- rychle bliká zelená LED1 (pravá LED na obrázku 18)
Startuje OS	- rychle blikají obě zelené LED
OS nastartován	- střídavě blikají (už pomaleji než v předchozím kroku) obě zelené LED
Přihlášení uživatele do OS a spuštění PC aplikace	- střídavě blikají obě zelené LED, pokud uživatel nezadá příkaz v PC aplikaci pro obsluhu LED - koncové zařízení reaguje na uživatelské příkazy

Tabulka 6: Chování koncového zařízení během zapínání počítače

7.5.2 Manipulace s USB kabelem

Následující odstavce shrnují chování spuštěné PC aplikace a koncového zařízení během manipulace s USB kabelem.

Vytažení USB kabelu koncového zařízení z PC

Koncové zařízení nepracuje protože ztratilo napájecí napětí. Spuštěná aplikace, která dosud komunikovala s koncovým zařízením vypíše následující:

```
Enter the command (h, q) or
Enter the USB device index followed by command (g, c, t, a, d, <0-9>):
Available USB device indexes: [0]
```

```
You have selected device 0, now enter the command:
Temperature: 25 °C
Error: callbackEventIn: status = 5, actual_length = 0
Error: Failed to submit EVENT IN transfer. -4
Error: callbackEventIn: Failed to resubmit event transfer. 1
```

PC aplikace nejdříve komunikovala správně s koncovým zařízením, přečetla teplotu – viz 5. řádek (`Temperature: 25 °C`). Potom došlo k vytažení USB kabelu z počítače a aplikace vypsala chybové zprávy. První řádek obsahuje status přenosu dat při obsluze událostí vysílaných koncovým zařízením: `status = 5`, což odpovídá hodnotě `LIBUSB_TRANSFER_NO_DEVICE` výčtového typu `libusb_transfer_status` knihovny LibUSB. To znamená, že PC aplikace se snaží komunikovat s koncovým zařízením, které není připojeno k počítači. Aplikace se pak snaží poslat požadavek pro obsluhu událostí, ale LibUSB vrací chybový kód `-4`, který odpovídá hodnotě `LIBUSB_ERROR_NO_DEVICE` výčtového typu `libusb_error` knihovny LibUSB. A to opět znamená, že PC aplikace se snaží komunikovat s koncovým zařízením, které není připojeno k počítači.

Zastrčení USB kabelu koncového zařízení do PC

Po následném zastrčení USB kabelu zpět do počítače, se zprovozní koncové zařízení, to už však nedokáže komunikovat s PC aplikací. Ta vypíše následující chybovou zprávu např. po zadání příkazu pro čtení teploty (uživatel stiskne klávesy `0` a `t`):

```
You have selected device 0, now enter the command:
Temerature: Write Failed with error code -4, transferred 0 bytes.
```

Což opět znamená, že PC aplikace se snaží komunikovat s koncovým zařízením, které není připojeno k počítači. Přesněji, knihovna LibUSB nezaregistrovala znovupřipojení koncového zařízení, protože nepodporuje funkcionalitu, která detekuje připojení a odpojení koncových zařízení za běhu aplikace. Pokud zavřeme a znovu spustíme PC aplikaci, komunikace s koncovým zařízením, jehož nastavení se vrátilo do výchozího stavu (zelené LED střídavě blikají, sedmisegmentový displej zobrazuje výchozí datum DD-MM-YYYY---), opět funguje bez problémů.

7.5.3 Odhlášení a přihlášení uživatele, zamčení obrazovky

Pokud se uživatel odhlásí z operačního systému, zavřou se všechny jeho spuštěné aplikace, takže koncové zařízení připojené k počítači funguje (stavy zelených LED a stav zobrazování data na sedmisegmentovém displeji jsou nezměněny), po stisknutí jakéhokoliv mikrospínače se však nic neděje. Když se uživatel přihlásí zpět, spustí PC aplikaci a zamkne obrazovku, koncové zařízení plně komunikuje s běžící aplikací.

7.5.4 Uspání a následné zapnutí počítače

Tabulka 7 shrnuje chování koncových zařízení během uspání a následném zapnutí počítače, po nastartování operačního systému Linux a následném spuštění PC aplikace.

Operace	Chování koncového zařízení (změny)
OS se vypíná	- rychle blikají obě zelené LED - sedmisegmentový displej zobrazuje poslední nastavené datum - po několika stovkách milisekund střídavě blikají (už pomaleji než v předchozím kroku) obě zelené LED
PC je uspán	- rychle blikají obě zelené LED
Zapnutí PC	- svítí zelená LED2 (levá LED na obrázku 18) - po několika stovkách milisekund střídavě blikají obě zelené LED - po několika stovkách milisekund svítí zelená LED1 (pravá LED na obrázku 18)
Startuje OS	- rychle blikají obě zelené LED
OS nastartován	- střídavě blikají (už pomaleji než v předchozím kroku) obě zelené LED
Přihlášení uživatele do OS a spuštění PC aplikace	- střídavě blikají obě zelené LED, pokud uživatel nezadá příkaz v PC aplikaci pro obsluhu LED - všechna koncová zařízení reagují na uživatelské příkazy

Tabulka 7: Chování koncového zařízení během uspání a následném zapnutí počítače

Po přihlášení uživatele je stav PC aplikace následující:

```
libusb:warning [handle_bulk_completion] unrecognised urb status -108
Error: callbackEventIn: status = 1, actual_length = 0
libusb:warning [handle_bulk_completion] unrecognised urb status -108
Error: callbackEventIn: status = 1, actual_length = 0
 --- Device R-A has sent the ASYNC EVENT !!!
```

Knihovna LibUSB vypsala v průběhu přenosu dat varování (první řádek: `status -108`), o kterém se mi nepodařilo zjistit bližší informace. V každém případě na dalším řádku je chybová hláška: `status = 1`, což odpovídá hodnotě `LIBUSB_TRANSFER_ERROR` výčtového typu `libusb_transfer_status` knihovny LibUSB. To znamená, že přenos dat mezi PC aplikace a koncovým zařízením selhal. Koncová zařízení však nadále komunikují s PC aplikací, jak dokazuje následující výpis aplikace:

```
 --- Device R-A has sent the ASYNC EVENT !!!
 --- Device R-A has sent the ASYNC EVENT !!!
```

```
The USB device requested to get a current date. It was successfuly sent.
You have selected device 0, now enter the command:
Temperature: 26 °C
The USB device requested to get a current date. It was successfuly sent.
You have selected device 1, now enter the command:
Temperature: 24 °C
```

7.5.5 Vypnutí a následné zapnutí počítače

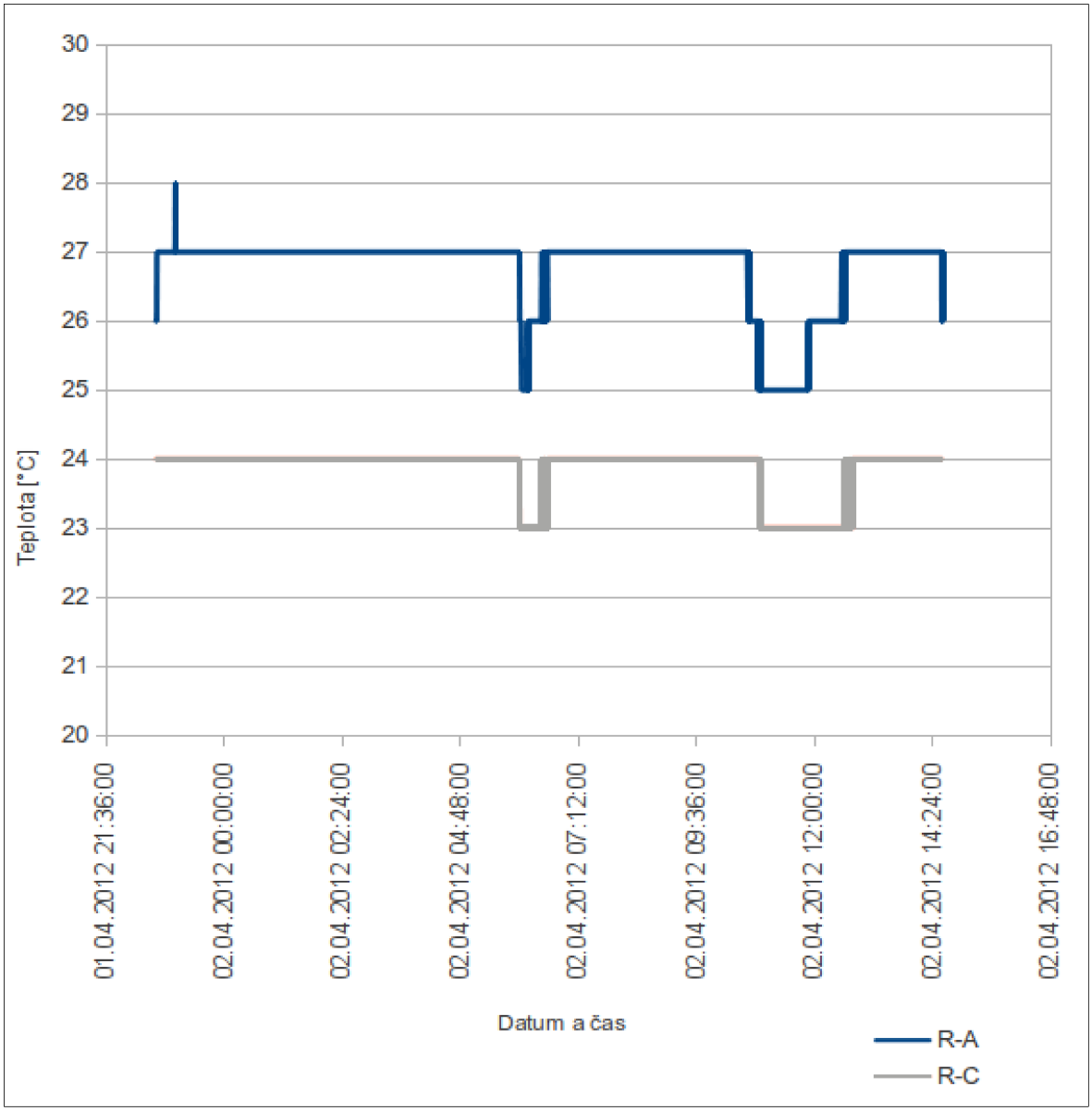
Tabulka 8 shrnuje chování koncových zařízení během vypnutí a následném zapnutí počítače, po nastartování operačního systému Linux a následném spuštění PC aplikace.

Operace	Chování koncového zařízení (změny)
Vypnutí PC aplikace	- střídavě blikají obě zelené LED - sedmisegmentový displej zobrazuje poslední nastavené datum
Vypnutí OS a následně PC	- rychle blikají obě zelené LED
Zapnutí PC	- svítí zelená LED2 (levá LED na obrázku 18)
Startuje BIOS	- rychle bliká zelená LED1 (pravá LED na obrázku 18)
Startuje OS	- rychle blikají obě zelené LED
OS nastartován	- střídavě blikají (už pomaleji než v předchozím kroku) obě zelené LED
Přihlášení uživatele do OS a spuštění PC aplikace	- střídavě blikají obě zelené LED, pokud uživatel nezadá příkaz v PC aplikaci pro obsluhu LED - všechna koncová zařízení reagují na uživatelské příkazy

Tabulka 8: Chování koncového zařízení během vypnutí a následném zapnutí počítače

7.5.6 Dlouhodobý provoz koncových zařízení

Nechal jsem PC aplikaci spuštěnou několik hodin. Z obou připojených koncových zařízení byla čtena teplota a zaznamenávána do datových CSV souborů pomocí třídy `Logger`. Každý soubor obsahoval datum, čas a hodnotu teploty jednoho zařízení. Z výsledných hodnot jsem sestrojil graf, který zobrazuje změnu teploty v závislosti na čase (viz obrázek 20). PC aplikace fungovala bez chyby stejně jako obě připojená USB koncová zařízení. Zvídavý čtenář si jistě všimne rozdílu naměřených teplot obou zařízení. Ten je zapřičiněn jiným použitým teplotním senzorem každého zařízení (R-A používá DS1621 a R-C používá ADT75) a umístěním koncového zařízení kolem počítače.



Obrázek 20: Graf závislosti teploty na čase

7.6 Elektrické vlastnosti

Tabulka 14 v příloze E shrnuje výsledky měření napájecího napětí a proudu na USB konektoru koncového zařízení. Můžeme z ní vyčíst, že na spotřebu proudu mají největší vliv svítivé diody. Celé zařízení odebírá proud minimálně 16,8 mA a maximálně 46,9 mA, což dává příkon 82,7 až 229,3 mW, v průměru tedy 156,8 mW.

8 Závěr

Úspěšně se mi podařilo vyzkoušet a ověřit funkcionalitu volně dostupné a šiřitelné knihovny LibUSB. Navrhl jsem a vytvořil demonstrační aplikaci, která s pomocí knihovny LibUSB umožňuje obousměrnou komunikaci mezi počítačem a mikrokontrolérem PIC18F26J50 přes USB rozhraní způsobem dotaz-odpověď (synchronně) a asynchronně. Napsal jsem také program pro mikrokontrolér (firmware), který obsluhuje USB rozhraní a další jednodušší periferie (LED, mikrospínače, sedmisegmentový displej, teplotní čidlo), díky němuž může PC aplikace komunikovat s mikrokontrolérem. Otestoval jsem aplikaci a koncová zařízení z pohledu stability, spolehlivosti, rychlosti, odezvy i komunikaci s více mikrokontroléry najednou.

PC aplikaci jsem napsal v programovacím jazyce C++ a firmware mikrokontroléru v programovacím jazyce C. Pro tvorbu schémat zapojení jsem použil aplikaci gschem a desku plošných spojů jsem navrhl pomocí aplikace pcb. Oba tyto volně šiřitelné programy jsou součástí projektu gEDA (viz literatura [18]) a lze je nainstalovat z databáze balíků operačního systému Ubuntu Linux.

Diplomová práce navíc popisuje vlastnosti mikrokontroléru PIC řady 18, jaké možnosti a vybavení nabízí dnešní mikrokontroléry a jaké prostředky můžeme použít při oživování mikrokontroléru. Obsahuje také informace o USB rozhraní, jak vypadá topologie rozhraní a jakým způsobem je komunikační systém rozdělen na vrstvy. Nastínil jsem základní zapojení mikrokontroléru PIC18F26J50 a návrh programového vybavení v jazyce C pro mikrokontrolér tzv. firmware. Popsal jsem, jak vytvořit jednoduchý program a jak jej přemístit na mikrokontrolér. Všechny tyto informace lze použít, jako doprovodný materiál při výuce.

Demonstrační PC aplikace ani firmware ani zapojení mikrokontroléru nejsou příliš složité, nepřináší žádné zásadní inovace v elektronickém průmyslu či v informačních technologiích, ale velice dobře ukazují, jak jednoduše a levně lze postavit malé zařízení, které dokáže komunikovat s počítačem přes USB rozhraní. V dnešní bezdrátové době bychom mohli rozšířit využití mikrokontroléru ke komunikaci přes rádiové vlny, např. pomocí technologie ZigBee, nebo s využitím jiných radiofrekvenčních (RF) modulů atd.

9 Literatura

[1] *Jednočipový počítač – Wikipedie* [online]. c2006, poslední revize 2011-07-14 [cit. 2012-02-25]. <http://cs.wikipedia.org/wiki/Jedno%C4%8Dipov%C3%BD_po%C4%8D%C3%ADta%C4%8D>.

[2] *Mikrokontroléry PIC – vše o osmibitových mikrokontrolérech PIC* [online]. c2012, poslední revize 2012-02-25 [cit. 2012-02-25]. <http://mikrokontrolery-pic.cz/>.

[3] Microchip Technology Inc. *Microchip C Compilers* [online]. c2011, poslední revize 2012-03-03 [cit. 2012-03-03]. <http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en534868&redirects=compilers>.

[4] Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V. *Universal Serial Bus Specification* [online]. c2000, poslední revize 2000-04-27 [cit. 2012-03-06]. <http://www.usb.org/developers/docs/usb_20_101111.zip> (usb_20.pdf).

[5] *libusb* [online]. c2009, poslední revize 2012-03-03 [cit. 2012-03-11]. <http://libusb.org>.

[6] *libusb: libusb-1.0 API Reference* [online]. c2010, poslední revize 2010-04-19 [cit. 2010-03-11]. <http://libusb.sourceforge.net/api-1.0/index.html>.

[7] *Xiaofan's Blog: libusb related wrappers or bindings* [online]. c2009, poslední revize 2009-05-03 [cit. 2012-03-17]. <http://mcuee.blogspot.com/2009/05/libusb-related-wrappers-or-bindings.html>.

[8] *openusb* [online]. c2008, poslední revize 2011-05-20 [cit. 2012-03-17]. <http://sourceforge.net/apps/trac/openusb>.

[9] *Xiaofan's Blog: OpenUSB is alive again* [online]. c2011, poslední revize 2011-04-28 [cit. 2012-03-17]. <http://mcuee.blogspot.com/2011/04/openusb-is-alive-again.html>.

[10] Microchip Technology Inc. *PIC18F46J50 Data Sheet* [online]. c2011, poslední revize 2011-04-05 [cit. 2012-03-17]. <http://ww1.microchip.com/downloads/en/DeviceDoc/39931d.pdf>. ISBN: 978-1-61341-027-1

[11] Analog Devices, Inc. *±1°C Accurate, 12-Bit Digital Temperature Sensor ADT75* [online]. c2005, poslední revize 2010-09-?? [cit. 2012-03-19]. <http://www.analog.com/static/imported-files/data_sheets/ADT75.pdf>.

[12] Microchip Technology Inc. *Getting Started with Development Tools* [online]. c1998-2012, poslední revize 2012-03-20 [cit. 2012-03-20]. <http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2123&param=en024282>.

[13] Microchip Technology Inc. *MPLAB C Compiler for PIC18 MCUs* [online]. c1998-2012, poslední revize 2012-03-20 [cit. 2012-03-20]. <http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010014>.

[14] ASIX s.r.o. *ASIX: PRESTO* [online]. c1991-2010, poslední revize 2010-12-10 [cit. 2012-03-22]. <http://asix.cz/prg_presto.htm>.

[15] Microchip Technology Inc. *Microchip Application Libraries* [online]. c1998-2012, poslední revize 2012-02-15 [cit. 2012-03-24]. <http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2680&dDocName=en547784>.

[16] Maxim Integrated Products. *DS1621 Digital Thermometer and Thermostat* [online]. c2005, poslední revize 2005-09-15 [cit. 2012-03-24]. <http://datasheets.maxim-ic.com/en/ds/DS1621.pdf>.

[17] Martin Olejár. *elweb.cz : fotocesta: domácí výroba desek plošných spojů (DPS) (101) : elektronika na webu Martina Olejára ::.* [online]. c1999-2012, poslední revize 2012-03-29 [cit. 2012-03-29]. <http://www.elweb.cz/clanky.php?clanek=101>.

[18] *gEDA Project's Homepage* [online]. c2012, poslední revize 2012-02-25 [cit. 2012-04-02]. <http://www.geda-project.org/>.

10 Přílohy

A Seznam potřebných souborů z frameworku MCHPFSUSB

Aplikační knihovny jsou po výchozí instalaci umístěny v následujícím adresáři:

```
~/microchip_solutions_v2011-07-14-beta
```

Následující soubory je nutné zkopírovat z adresáře aplikačních knihoven do vlastního projektu pro správnou funkci USB modulu:

- `Microchip/Include/Compiler.h`
- `Microchip/Include/GenericTypeDefs.h`
- `Microchip/Include/USB/usb.h`
- `Microchip/Include/USB/usb_ch9.h`
- `Microchip/Include/USB/usb_common.h`
- `Microchip/Include/USB/usb_device.h`
- `Microchip/Include/USB/usb_function_generic.h`
- `Microchip/Include/USB/usb_hal.h`
- `Microchip/Include/USB/usb_hal_pic18.h`
- `Microchip/USB/usb_device.c`
- `Microchip/USB/usb_device_local.h`
- `Microchip/USB/Generic Device Driver/usb_function_generic.c`
- `USB/Device – LibUSB – Generic Driver Demo/Firmware/HardwareProfile.h`
- `USB/Device – LibUSB – Generic Driver Demo/Firmware/HardwareProfile - PIC18F46J50 PIM.h`
- `USB/Device – LibUSB – Generic Driver Demo/Firmware/main.c`
- `USB/Device – LibUSB – Generic Driver Demo/Firmware/usb_config.h`
- `USB/Device – LibUSB – Generic Driver Demo/Firmware/usb_descriptors.c`

B Převod znaku (čísla) sedmisegmentového displeje na hodnotu portu A

Znak zobrazitelný na sedmisegmentovém displeji	Hodnota portu A (bit 7, 6, 5, 4, 3, 2, 1, 0), nebo segmenty (g, f, e, -, d, c, b, a)	Kód znaku (hexadecimální vyjádření hodnoty portu A)
0	1000 0000	0x80
1	1110 1001	0xE9
2	0100 0100	0x44
3	0110 0000	0x60
4	0010 1001	0x29
5	0010 0010	0x22
6	0000 0010	0x02
7	1110 1000	0xE8
8	0000 0000	0x00
9	0010 0000	0x20
A	0000 1000	0x08
b	0000 0011	0x03
C	1000 0110	0x86
d	0100 0001	0x41
E	0000 0110	0x06
F	0000 1110	0x0E
H	0000 1001	0x09
I	1000 1111	0x8F
J	1100 0001	0xC1
L	1000 0111	0x87
M	1000 1000	0x88
O	1000 0000	0x80
P	0000 1100	0x0C
S	0010 0010	0x22
U	1000 0001	0x81
-	0110 1111	0x6F
Symbol chyby ≡	0110 0110	0x66

Tabulka 9: Převod znaku (čísla) sedmisegmentového displeje na hodnotu portu A

C Naměřené hodnoty doby přenosu dat z jednoho zařízení komunikujícího s PC aplikací

První řádek označuje typ rozhraní (synchronní/asynchronní), 2. řádek hlavičky obsahuje typ operace, 3. řádek označuje směr přenosu dat (OUT = z kořenového uzlu na koncové zařízení, IN = z koncového zařízení na kořenový uzel). Další řádky obsahují doby přenosu dat pro jednotlivá měření a poslední 3 řádky jsou vypočtené minimální, maximální a průměrné hodnoty časů.

Rozhraní:	**Synchronní přenos dat**								**Asynchronní přenos dat**		
Operace:	**Čtení teploty**		**Čtení ID**		**Čtení dat**		**Zápis data**		**Čtení teploty**		**Událost**
Směr:	**OUT**	**IN**	**OUT**	**IN**	**OUT**	**IN**	**IN**	**OUT**	**OUT**	**IN**	**IN**
Čas [µs]:	778	949	771	947	788	928	915	948	696	959	779
Čas [µs]:	794	815	781	942	784	938	937	946	597	785	763
Čas [µs]:	779	947	760	948	778	945	927	950	709	859	800
Čas [µs]:	774	937	786	949	802	940	942	944	708	851	769
Čas [µs]:	759	946	790	943	788	950	937	949	705	849	779
Čas [µs]:	769	946	773	939	745	947	936	941	653	798	782
Čas [µs]:	765	943	765	946	775	949	939	956	705	849	783
Čas [µs]:	785	915	814	405	763	950	931	1 066	702	846	780
Čas [µs]:	785	946	774	946	778	941	710	902	682	826	787
Čas [µs]:	734	948	781	949	763	947	909	937	635	815	787
Čas [µs]:	776	952	771	946	783	914	918	944	714	854	782
Čas [µs]:	771	920	766	938	785	951	917	949	712	847	780
Čas [µs]:	771	948	760	923	773	971	877	950	718	856	755
Čas [µs]:	759	946	770	938	783	948	919	944	713	850	785
Čas [µs]:	786	948	769	945	752	933	923	950	715	857	778
Čas [µs]:	760	948	769	946	778	950	925	992	651	797	782
Čas [µs]:	765	929	768	950	785	929	779	937	711	848	752
Čas [µs]:	763	946	772	931	772	948	930	944	680	817	817
Čas [µs]:	769	952	763	954	758	947	938	955	711	851	781
Čas [µs]:	761	947	772	951	785	820	921	947	710	846	785
Min:	734	815	760	405	745	820	710	902	597	785	752
Max:	794	952	814	954	802	971	942	1 066	718	959	817
Průměr:	770	936	774	917	776	937	907	953	691	843	780

Tabulka 10: Rychlost přenosu dat – koncové zařízení na nepájivém poli

Rozhraní:	Synchronní přenos dat								Asynchronní přenos dat		
Operace:	Čtení teploty		Čtení ID		Čtení dat		Zápis data		Čtení teploty		Událost
Směr:	OUT	IN	OUT	IN	OUT	IN	IN	OUT	OUT	IN	IN
Čas [µs]:	814	845	748	950	768	949	908	945	646	1 721	776
Čas [µs]:	789	958	764	933	766	949	938	940	688	1 764	774
Čas [µs]:	793	947	773	945	742	943	940	917	682	828	770
Čas [µs]:	784	946	766	947	781	945	935	952	694	1 777	774
Čas [µs]:	770	955	771	938	783	921	939	949	697	1 774	774
Čas [µs]:	776	950	764	947	776	945	932	949	707	851	779
Čas [µs]:	748	946	767	940	787	968	945	944	691	1 769	776
Čas [µs]:	761	951	761	947	763	953	934	958	701	1 771	777
Čas [µs]:	767	842	745	947	772	949	914	954	693	1 772	771
Čas [µs]:	802	846	763	943	859	974	904	950	650	1 766	783
Čas [µs]:	795	813	755	947	773	949	916	949	692	1 765	783
Čas [µs]:	773	954	761	942	772	952	915	936	713	854	778
Čas [µs]:	758	946	826	831	795	834	915	950	720	857	772
Čas [µs]:	805	851	773	946	763	951	912	951	717	852	727
Čas [µs]:	796	822	792	942	792	949	922	949	701	1 803	779
Čas [µs]:	785	937	777	948	773	949	919	1001	708	848	777
Čas [µs]:	766	947	800	941	770	952	769	938	714	851	776
Čas [µs]:	783	934	786	948	787	939	904	948	708	847	782
Čas [µs]:	779	983	774	927	751	918	939	956	692	1 766	791
Čas [µs]:	790	942	816	834	771	940	937	950	677	1 751	715
Min:	748	813	745	831	742	834	769	917	646	828	715
Max:	814	983	826	950	859	974	945	1001	720	1 803	791
Průměr:	782	916	774	932	777	941	917	949	695	1 399	772

Tabulka 11: Rychlost přenosu dat – koncové zařízení na desce plošných spojů

D Naměřené hodnoty doby přenosu dat z dvou zařízení komunikujících s PC aplikací současně

Rozhraní:	**Synchronní přenos dat**				**Asynchronní přenos dat**		
Operace:	**Čtení teploty**		**Zápis data**		**Čtení teploty**		**Událost**
Směr:	**OUT**	**IN**	**IN**	**OUT**	**OUT**	**IN**	**IN**
Čas [µs]:	775	940	948	936	644	795	858
Čas [µs]:	787	951	949	942	703	847	722
Čas [µs]:	769	942	943	940	705	853	715
Čas [µs]:	781	949	957	932	704	846	788
Čas [µs]:	824	942	952	950	705	850	715
Čas [µs]:	773	942	947	942	695	836	719
Čas [µs]:	741	949	833	921	687	836	716
Čas [µs]:	763	948	928	936	700	843	686
Čas [µs]:	778	951	943	948	647	844	717
Čas [µs]:	769	949	949	945	666	846	694
Čas [µs]:	772	941	852	908	664	803	721
Čas [µs]:	786	948	937	944	672	813	678
Čas [µs]:	772	946	941	944	697	831	710
Čas [µs]:	779	946	945	946	706	844	723
Čas [µs]:	773	940	922	959	703	839	729
Čas [µs]:	768	951	286	787	701	838	718
Čas [µs]:	760	949	946	942	698	834	726
Čas [µs]:	779	954	931	948	706	842	727
Čas [µs]:	772	950	930	945	708	852	731
Čas [µs]:	777	927	928	1 226	714	871	660
Min:	741	927	286	787	644	795	660
Max:	824	954	957	1 226	714	871	858
Průměr:	775	946	898	947	691	838	723

Tabulka 12: Rychlost přenosu dat – koncové zařízení na nepájivém poli (2 zařízení komunikující současně)

Rozhraní:	Synchronní přenos dat				Asynchronní přenos dat		
Operace:	Čtení teploty		Zápis data		Čtení teploty		Událost
Směr:	OUT	IN	IN	OUT	OUT	IN	IN
Čas [µs]:	777	943	904	907	671	1 752	722
Čas [µs]:	773	944	942	938	670	1 743	666
Čas [µs]:	766	948	945	944	707	1 780	713
Čas [µs]:	767	954	940	939	690	1 766	709
Čas [µs]:	765	945	936	942	697	1 773	706
Čas [µs]:	771	942	917	935	692	1 769	682
Čas [µs]:	742	921	932	970	703	847	711
Čas [µs]:	772	938	922	878	687	1 763	848
Čas [µs]:	757	925	840	942	713	853	708
Čas [µs]:	769	935	944	943	651	1 753	673
Čas [µs]:	773	922	899	942	704	1 776	711
Čas [µs]:	758	942	923	936	712	847	711
Čas [µs]:	642	618	924	941	694	1 768	703
Čas [µs]:	765	942	943	934	712	848	700
Čas [µs]:	761	944	988	774	706	1 776	720
Čas [µs]:	728	942	891	4 447	671	1 742	1 012
Čas [µs]:	791	945	906	951	717	1 781	697
Čas [µs]:	772	944	943	955	703	1 773	669
	770	940	950	943	839	907	709
Čas [µs]:	769	945	929	955	680	1 748	856
Min:	642	618	840	774	651	847	666
Max:	791	954	988	4 447	839	1 781	1 012
Průměr:	759	924	926	1 106	701	1 538	731

Tabulka 13: Rychlost přenosu dat – koncové zařízení na desce plošných spojů (2 zařízení komunikující současně)

E Naměřené hodnoty napětí a proudu v závislosti na prováděné činnosti zařízení

Předpoklad měření	Napětí [V]		Proud [mA]		Výkon [mW]	
	MIN	MAX	MIN	MAX	MIN	MAX
Zařízení je připojeno, nekomunikuje s PC aplikací	4,89	4,92	18,2	41,5	89,0	204,2
Zařízení komunikuje s PC aplikací – blikají obě LED	4,89	4,92	20,3	44,4	99,3	218,4
Zařízení komunikuje s PC aplikací – nesvítí LED	4,92	4,92	18,3	22,7	90,0	111,7
Zařízení komunikuje s PC aplikací – svítí obě LED	4,89	4,89	42	46,5	205,4	227,4
Automatické čtení teploty – blikají obě LED	4,89	4,92	20,9	44,3	102,2	218,0
Automatické čtení teploty – nesvítí LED	4,92	4,92	18,4	22,8	90,5	112,2
Automatické čtení teploty – svítí obě LED	4,89	4,89	42	46,5	205,4	227,4
Zařízení posílá událost alarm – blikají obě LED	4,89	4,92	18,4	45,1	90,0	221,9
Zařízení posílá událost alarm – nesvítí LED	4,92	4,92	16,8	23,3	82,7	114,6
Zařízení posílá událost alarm – svítí obě LED	4,89	4,89	40,3	46,9	197,1	229,3
Min	4,89	4,89	16,8	22,7	82,7	111,7
Max	4,92	4,92	42	46,9	205,4	229,3
Průměr	4,90	4,91	25,6	38,4	125,1	188,5
Průměr min. i max. hodnot	4,91		32,0		156,8	

Tabulka 14: Hodnoty napětí a proudu v závislosti na prováděné činnosti zařízení

F Příloha na CD

- kopie tohoto dokumentu psaného v aplikaci LibreOffice 3.4.4
- zdrojové kódy PC aplikace
- zdrojové kódy programu pro mikrokontrolér (firmware)
- projekty aplikací gschem a pcb obsahující schéma zapojení a návrh desky plošných spojů
- projekt aplikace BOUML obsahující diagramy komponent a třídní diagramy
- fotky, obrázky a výpisy aplikace použité v tomto dokumentu